Sparse Arrays

Julia has support for sparse vectors and sparse matrices in the `SparseArrays` stdlib module. Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

Compressed Sparse Column (CSC) Sparse Matrix Storage

In Julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC{Tv,Ti}`, where `Tv` is the type of the stored values, and `Ti` is the integer type for storing column pointers and row indices. The internal representation of `SparseMatrixCSC` is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of previously unstored entries one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted. If your `SparseMatrixCSC` object contains unsorted row indices, one quick way to sort them is by doing a double transpose.

In some applications, it is convenient to store explicit zero values in a `SparseMatrixCSC`. These *are* accepted by functions in `Base` (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The `nnz` function returns the number of elements explicitly stored in the sparse data structure, including structural nonzeros. In order to count the exact number of numerical nonzeros, use `count(!iszero, x)`, which inspects every stored element of a sparse matrix. `dropzeros`, and the in-place `dropzeros!`, can be used to remove stored zeros from the sparse matrix.

```
julia> A = sparse([1, 1, 2, 3], [1, 3, 2, 3], [0, 1, 2, 0])
3×3 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1, 1]  =  0
  [2, 2]  =  2
  [1, 3]  =  1
  [3, 3]  =  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64,Int64} with 2 stored entries:
  [2, 2]  =  2
  [1, 3]  =  1
```

Sparse Vector Storage

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type `SparseVector{Tv,Ti}` where `Tv` is the type of the stored values and `Ti` the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int              # Length of the sparse vector
    nzind::Vector{Ti}   # Indices of stored values
    nzval::Vector{Tv}   # Stored values, typically nonzeros
end
```

As for `SparseMatrixCSC`, the `SparseVector` type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

Sparse Vector and Matrix Constructors

The simplest way to create a sparse array is to use a function equivalent to the `zeros` function that Julia provides for working with dense arrays. To produce a sparse array instead, you can use the same name with an `sp` prefix:

```
julia> spzeros(3)
3-element SparseVector{Float64,Int64} with 0 stored entries
```

The `sparse` function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of stored values (this is also known as the COO (coordinate) format). `sparse(I,J,V)` then constructs a sparse matrix such that `S[I[k], J[k]] = V[k]`. The equivalent sparse vector constructor is `sparsevec`, which takes the (row) index vector `I` and the vector `V` with the stored values and constructs a sparse vector `R` such that `R[I[k]] = V[k]`.

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1,  4]  =  1
  [4,  7]  =  2
  [5,  9]  =  3
  [3, 18]  =  -5

julia> R = sparsevec(I,V)
5-element SparseVector{Int64,Int64} with 4 stored entries:
  [1]  =  1
  [3]  =  -5
  [4]  =  2
  [5]  =  3
```

The inverse of the `sparse` and `sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array. `findall(!iszero, x)` returns the cartesian indices of non-zero entries in `x` (including stored entries equal to zero).

```
julia> findnz(S)
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])

julia> findall(!iszero, S)
4-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 4)
 CartesianIndex(4, 7)
 CartesianIndex(5, 9)
 CartesianIndex(3, 18)

julia> findnz(R)
([1, 3, 4, 5], [1, -5, 2, 3])
```

```
julia> findall(!iszero, R)
4-element Array{Int64,1}:
 1
 3
 4
 5
```

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

```
julia> sparse(Matrix(1.0I, 5, 5))
5×5 SparseMatrixCSC{Float64,Int64} with 5 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  1.0
  [5, 5]  =  1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

You can go in the other direction using the `Array` constructor. The `issparse` function can be used to query if a matrix is sparse.

```
julia> issparse(spzeros(5))
true
```

Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into `(I,J,V)` format using `findnz`, manipulate the values or the structure in the dense vectors `(I,J,V)`, and then reconstruct the sparse matrix.

Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a given sparse matrix `S`, or that the resulting sparse matrix has density `d`, i.e. each matrix element has a probability `d` of being non-zero.

Details can be found in the Sparse Vectors and Matrices section of the standard library reference.

Sparse	Dense	Description
`spzeros(m,n)`	`zeros(m,n)`	Creates a *m*-by-*n* matrix of zeros. (`spzeros(m,n)` is empty.)
`sparse(I, n, n)`	`Matrix(I,n,n)`	Creates a *n*-by-*n* identity matrix.
`Array(S)`	`sparse(A)`	Interconverts between dense and sparse formats.
`sprand(m,n,d)`	`rand(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
`sprandn(m,n,d)`	`randn(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
`sprandn(rng,m,n,d)`	`randn(rng,m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements generated with the `rng` random number generator

Sparse Arrays

`SparseArrays.AbstractSparseArray` — Type

```
AbstractSparseArray{Tv,Ti,N}
```

Supertype for `N`-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. `SparseMatrixCSC`, `SparseVector` and `SuiteSparse.CHOLMOD.Sparse` are subtypes of this.

`SparseArrays.AbstractSparseVector` — Type

```
AbstractSparseVector{Tv,Ti}
```

Supertype for one-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. Alias for `AbstractSparseArray{Tv,Ti,1}`.

`SparseArrays.AbstractSparseMatrix` — Type

```
AbstractSparseMatrix{Tv,Ti}
```

Supertype for two-dimensional sparse arrays (or array-like types) with elements of type `Tv` and index type `Ti`. Alias for `AbstractSparseArray{Tv,Ti,2}`.

`SparseArrays.SparseVector` — Type

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

`SparseArrays.SparseMatrixCSC` — Type

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format. The standard way of constructing SparseMatrixCSC is through the `sparse` function. See also `spzeros`, `spdiagm` and `sprand`.

`SparseArrays.sparse` — Function

```
sparse(A)
```

Convert an AbstractMatrix `A` into a sparse matrix.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> sparse(A)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
```

```
sparse(I, J, V,[ m, n, combine])
```

Create a sparse matrix `S` of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of `V` are Booleans in which case `combine` defaults to `|`. All elements of `I` must satisfy `1 <= I[k] <= m`, and all elements of `J` must satisfy `1 <= J[k] <= n`. Numerical zeros in (`I`, `J`, `V`) are retained as structural nonzeros; to drop numerical zeros, use `dropzeros!`.

For additional documentation and an expert driver, see `SparseArrays.sparse!`.

Examples

```
julia> Is = [1; 2; 3];

julia> Js = [1; 2; 3];

julia> Vs = [1; 2; 3];

julia> sparse(Is, Js, Vs)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  1
  [2, 2]  =  2
  [3, 3]  =  3
```

`SparseArrays.sparsevec` — Function

```
sparsevec(I, V, [m, combine])
```

Create a sparse vector `S` of length `m` such that `S[I[k]] = V[k]`. Duplicates are combined using the `combine` function, which defaults to `+` if no `combine` argument is provided, unless the elements of `V` are Booleans in which case `combine` defaults to `|`.

Examples

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];

julia> sparsevec(II, V)
5-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  0.5
  [5]  =  0.2

julia> sparsevec(II, V, 8, -)
8-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  -0.1
  [5]  =  0.2

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false])
3-element SparseVector{Bool,Int64} with 3 stored entries:
  [1]  =  1
  [2]  =  0
  [3]  =  1
```

```
sparsevec(d::Dict, [m])
```

Create a sparse vector of length `m` where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

Examples

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64,Int64} with 2 stored entries:
  [1]  =  3
```

```
  [2]  =  2
```

```
sparsevec(A)
```

Convert a vector `A` into a sparse vector of length `m`.

Examples

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  2.0
  [5]  =  3.0
```

`SparseArrays.issparse` — Function

```
issparse(S)
```

Returns `true` if `S` is sparse, and `false` otherwise.

Examples

```
julia> sv = sparsevec([1, 4], [2.3, 2.2], 10)
10-element SparseVector{Float64,Int64} with 2 stored entries:
  [1 ]  =  2.3
  [4 ]  =  2.2

julia> issparse(sv)
true

julia> issparse(Array(sv))
false
```

`SparseArrays.nnz` — Function

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nnz(A)
3
```

`SparseArrays.findnz` — Function

```
findnz(A)
```

Return a tuple `(I, J, V)` where `I` and `J` are the row and column indices of the stored ("structurally non-zero") values in sparse matrix `A`, and `V` is a vector of the values.

Examples

```
julia> A = sparse([1 2 0; 0 0 3; 0 4 0])
3×3 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1, 1]  =  1
  [1, 2]  =  2
  [3, 2]  =  4
  [2, 3]  =  3

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

`SparseArrays.spzeros` — Function

```
spzeros([type,]m[,n])
```

Create a sparse vector of length `m` or sparse matrix of size `m x n`. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction.

The type defaults to `Float64` if not specified.

Examples

```
julia> spzeros(3, 3)
3×3 SparseMatrixCSC{Float64,Int64} with 0 stored entries

julia> spzeros(Float32, 4)
4-element SparseVector{Float32,Int64} with 0 stored entries
```

`SparseArrays.spdiagm` — Function

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
spdiagm(m::Integer, n::Ingeger, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a sparse diagonal matrix from `Pair`s of vectors and diagonals. Each vector `kv.second` will be placed on the `kv.first` diagonal. By default (if `size=nothing`), the matrix is square and its size is inferred from `kv`, but a non-square size `m`×`n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments.

Examples

```
julia> spdiagm(-1 => [1,2,3,4], 1 => [4,3,2,1])
5×5 SparseMatrixCSC{Int64,Int64} with 8 stored entries:
  [2, 1]  =  1
  [1, 2]  =  4
  [3, 2]  =  2
  [2, 3]  =  3
  [4, 3]  =  3
  [3, 4]  =  2
  [5, 4]  =  4
  [4, 5]  =  1
```

`SparseArrays.blockdiag` — Function

```
blockdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

```
julia> blockdiag(sparse(2I, 3, 3), sparse(4I, 2, 2))
5×5 SparseMatrixCSC{Int64,Int64} with 5 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2
  [4, 4]  =  4
  [5, 5]  =  4
```

SparseArrays.sprand — Function

```
sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

Create a random length `m` sparse vector or `m` by `n` sparse matrix, in which the probability of any element being nonzero is independently given by `p` (and hence the mean density of nonzeros is also exactly `p`). Nonzero values are sampled from the distribution specified by `rfn` and have the type `type`. The uniform distribution is used in case `rfn` is not specified. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> sprand(Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool,Int64} with 1 stored entry:
  [2, 2]  =  1

julia> sprand(Float64, 3, 0.75)
3-element SparseVector{Float64,Int64} with 1 stored entry:
  [3]  =  0.298614
```

SparseArrays.sprandn — Function

```
sprandn([rng][,Type],m[,n],p::AbstractFloat)
```

Create a random sparse vector of length `m` or sparse matrix of size `m` by `n` with the specified (independent) probability `p` of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional `rng` argument specifies a random number generator, see

Random Numbers.

Julia 1.1

Specifying the output element type `Type` requires at least Julia 1.1.

Examples

```
julia> sprandn(2, 2, 0.75)
2×2 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 2]  =  0.586617
  [2, 2]  =  0.297336
```

`SparseArrays.nonzeros` — Function

```
nonzeros(A)
```

Return a vector of the structural nonzero values in sparse array `A`. This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of `A`, and any modifications to the returned vector will mutate `A` as well. See `rowvals` and `nzrange`.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nonzeros(A)
3-element Array{Int64,1}:
 2
 2
 2
```

`SparseArrays.rowvals` — Function

```
rowvals(A::AbstractSparseMatrixCSC)
```

Return a vector of the row indices of `A`. Any modifications to the returned vector will mutate `A` as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also `nonzeros` and `nzrange`.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> rowvals(A)
3-element Array{Int64,1}:
 1
 2
 3
```

`SparseArrays.nzrange` — Function

```
nzrange(A::AbstractSparseMatrixCSC, col::Integer)
```

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with `nonzeros` and `rowvals`, this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for j = 1:n
   for i in nzrange(A, j)
      row = rows[i]
      val = vals[i]
      # perform sparse wizardry...
   end
end
```

`SparseArrays.droptol!` — Function

```
droptol!(A::AbstractSparseMatrixCSC, tol)
```

Removes stored values from `A` whose absolute value is less than or equal to `tol`.

```
droptol!(x::SparseVector, tol)
```

Removes stored values from `x` whose absolute value is less than or equal to `tol`.

`SparseArrays.dropzeros!` — Function

```
dropzeros!(A::AbstractSparseMatrixCSC;)
```

Removes stored numerical zeros from `A`.

For an out-of-place version, see `dropzeros`. For algorithmic information, see `fkeep!`.

```
dropzeros!(x::SparseVector)
```

Removes stored numerical zeros from `x`.

For an out-of-place version, see `dropzeros`. For algorithmic information, see `fkeep!`.

`SparseArrays.dropzeros` — Function

```
dropzeros(A::AbstractSparseMatrixCSC;)
```

Generates a copy of `A` and removes stored numerical zeros from that copy.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  0.0
  [3, 3]  =  1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  1.0
  [3, 3]  =  1.0
```

```
dropzeros(x::SparseVector)
```

Generates a copy of `x` and removes numerical zeros from that copy.

For an in-place version and algorithmic information, see `dropzeros!`.

Examples

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  0.0
```

```
  [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

`SparseArrays.permute` — Function

```
permute(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
        q::AbstractVector{<:Integer}) where {Tv,Ti}
```

Bilaterally permute `A`, returning `PAQ` (`A[p,q]`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == size(A, 2)`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == size(A, 1)`).

For expert drivers and additional information, see `permute!`.

Examples

```
julia> A = spdiagm(0 => [1, 2, 3, 4], 1 => [5, 6, 7])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [1, 1]  =  1
  [1, 2]  =  5
  [2, 2]  =  2
  [2, 3]  =  6
  [3, 3]  =  3
  [3, 4]  =  7
  [4, 4]  =  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [4, 1]  =  1
  [3, 2]  =  2
  [4, 2]  =  5
  [2, 3]  =  3
  [3, 3]  =  6
  [1, 4]  =  4
  [2, 4]  =  7

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
```

```
  [3, 1]  =  7
  [4, 1]  =  4
  [2, 2]  =  6
  [3, 2]  =  3
  [1, 3]  =  5
  [2, 3]  =  2
  [1, 4]  =  1
```

`Base.permute!` — Method

```
permute!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
         [C::AbstractSparseMatrixCSC{Tv,Ti}]) where {Tv,Ti}
```

Bilaterally permute `A`, storing result `PAQ` (`A[p,q]`) in `X`. Stores intermediate result `(AQ)^T` (`transpose(A[:,q])`) in optional argument `C` if present. Requires that none of `X`, `A`, and, if present, `C` alias each other; to store result `PAQ` back into `A`, use the following method lacking `X`:

```
permute!(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
         q::AbstractVector{<:Integer}[, C::AbstractSparseMatrixCSC{Tv,Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

`X`'s dimensions must match those of `A` (`size(X, 1) == size(A, 1)` and `size(X, 2) == size(A, 2)`), and `X` must have enough storage to accommodate all allocated entries in `A` (`length(rowvals(X)) >= nnz(A)` and `length(nonzeros(X)) >= nnz(A)`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == size(A, 2)`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == size(A, 1)`).

`C`'s dimensions must match those of `transpose(A)` (`size(C, 1) == size(A, 2)` and `size(C, 2) == size(A, 1)`), and `C` must have enough storage to accommodate all allocated entries in `A` (`length(rowvals(C)) >= nnz(A)` and `length(nonzeros(C)) >= nnz(A)`).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `unchecked_noalias_permute!` and `unchecked_aliasing_permute!`.

See also: `permute`.

« Sockets Statistics »