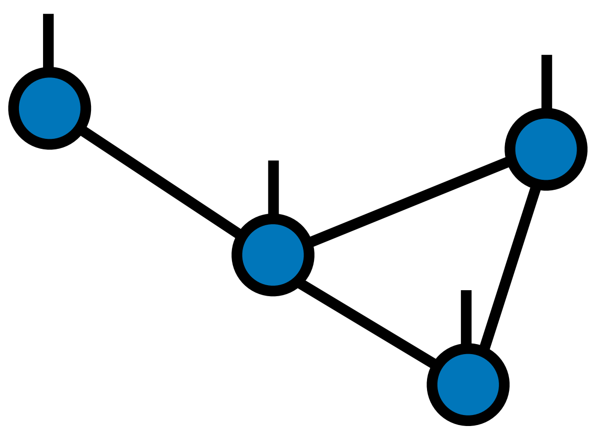
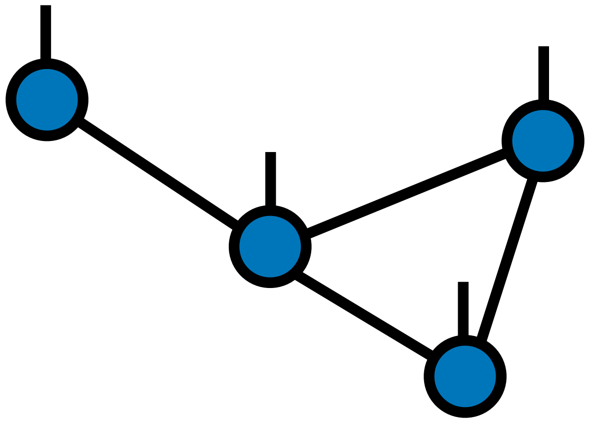
ITensors.jl Under the Hood



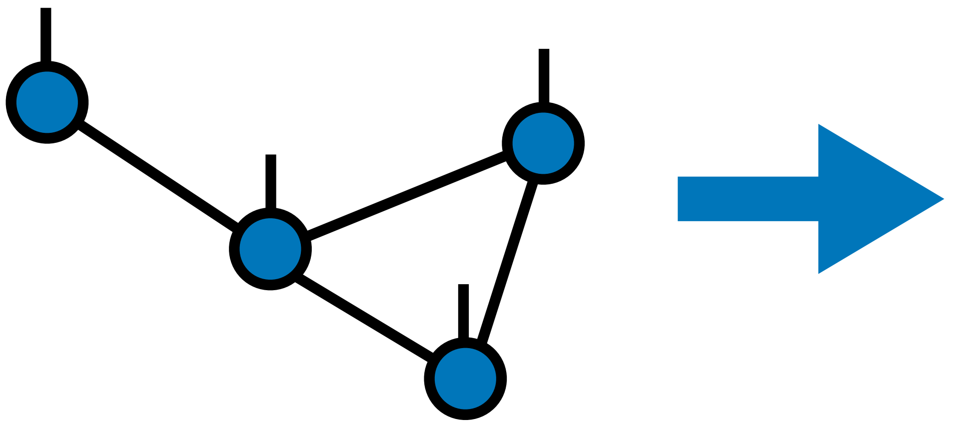
```julia
function hubbard_chain(sites; N, t=1.0, U=4.0)
    terms = OpSum()
    for j in 1:(N - 1)
        terms -= t, "Cdagup", j, "Cup", j+1
        terms -= t, "Cdagup", j+1, "Cup", j
        terms -= t, "Cdagdn", j, "Cdn", j+1
        terms -= t, "Cdagdn", j+1, "Cdn", j
    end
    [...]
end
```

Motivation

Tensor diagrams are a powerful notation for tensor networks and tensor contractions

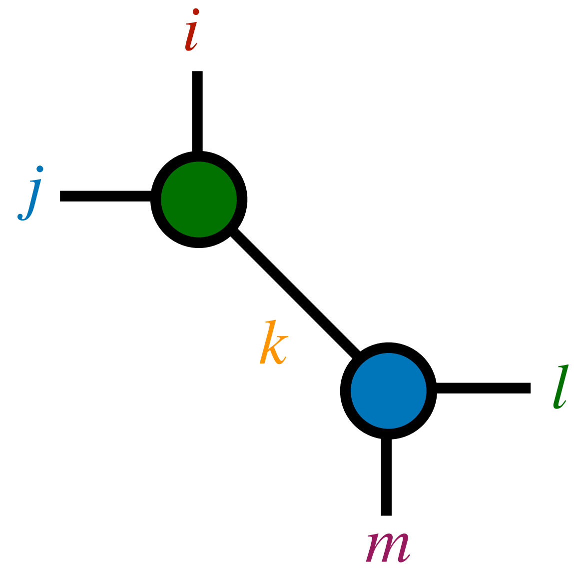


What if there was software modeled on tensor diagrams?



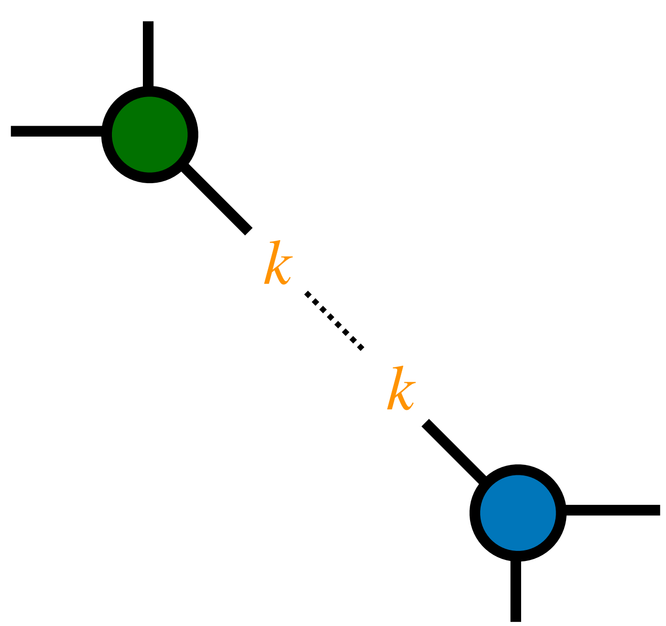
```julia
function hubbard_chain(sites; N, t=1.0, U=4.0)
    terms = OpSum()
    for j in 1:(N - 1)
      terms -= t, "Cdagup", j, "Cup", j+1
      terms -= t, "Cdagup", j+1, "Cup", j
      terms -= t, "Cdagdn", j, "Cdn", j+1
      terms -= t, "Cdagdn", j+1, "Cdn", j
    end
    [...]
end
```

Reminder: connecting index line means contraction

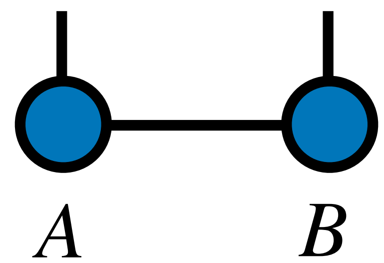


Which indices can connect?

Should be the **same** index – same size and <u>meaning</u>



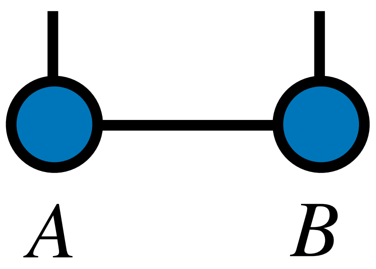
Some tensor libraries use index positions...




```
contract(A,{1,2},B,{2,3})
```

Must know contracted index is:
second index of A, first index of B

String identifiers help...




```
contract(A,{'i','j'},B,{'j','k'})
```

But the strings are temporary

Not unique – might repeat later

The ITensor Software

Can indices "remember" their identity?

Before making an ITensor, we make indices

```
julia> i = Index(3)
       (dim=3|id=807)
```

Before making an ITensor, we make indices

```
julia> i = Index(3)
       (dim=3|id=807)
```

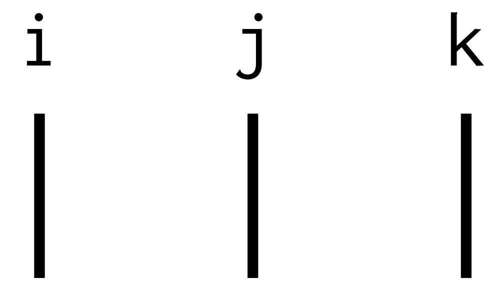
dimension unique identifier

Identifier lets index "recognize" itself

"Intelligent" index

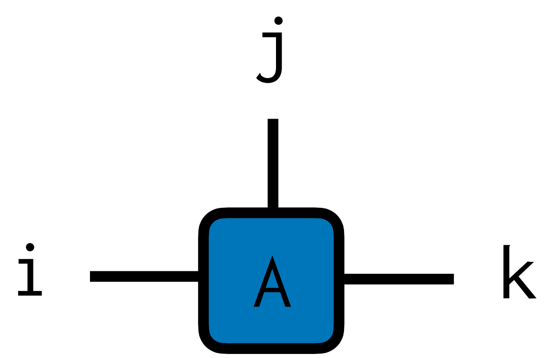
Making some indices

```
julia> i = Index(3)
julia> j = Index(2)
julia> k = Index(4)
```

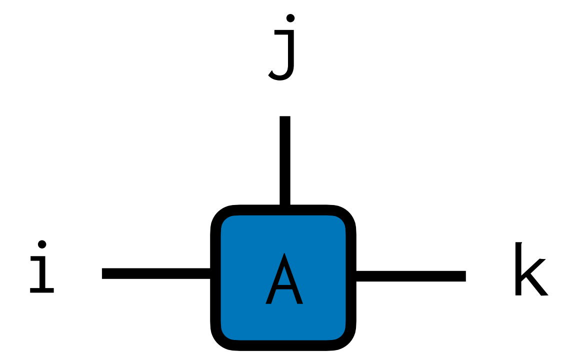


Now we can make tensors

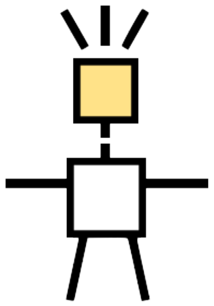
```
julia> A = ITensor(i,j,k)
```



```
julia> A = ITensor(i,j,k)
```



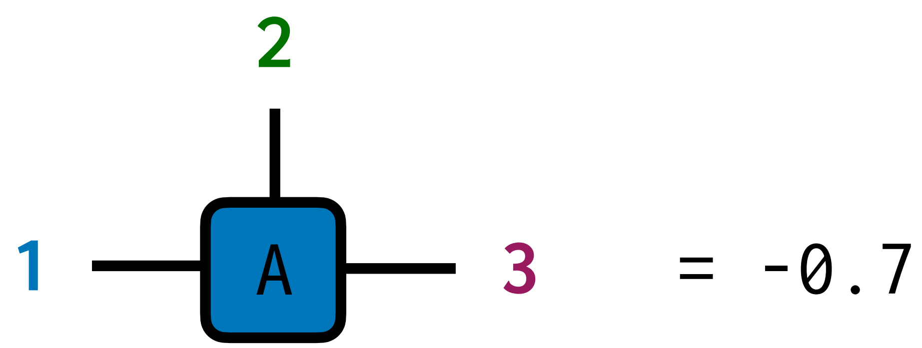
Indices are intelligent –
an intelligent tensor or **ITensor** *



* Not iTensor or i-Tensor … 🙂

Setting element of an ITensor

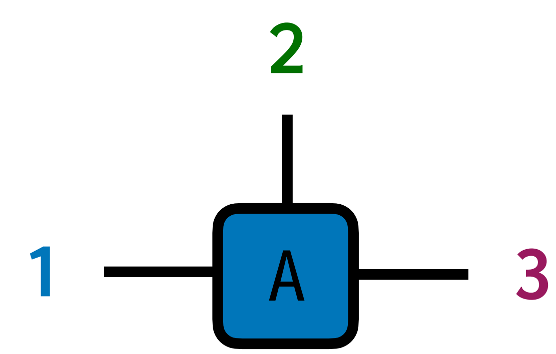
```
julia> A[i=>1,j=>2,k=>3] = -0.7
```



Any order allowed since indices known

```
julia> A[i=>1,j=>2,k=>3] = -0.7
```

```
julia> A[j=>2,i=>1,k=>3] = -0.7
```



= -0.7

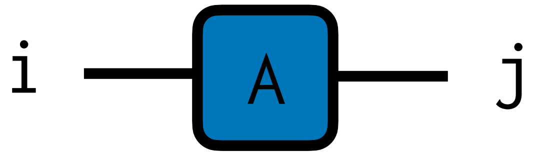
Both commands have same effect

Operations with ITensors

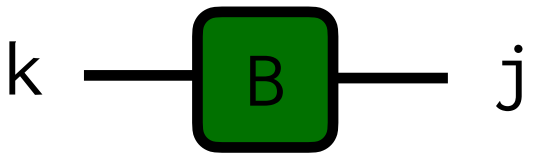
Contracting ITensors

Given two ITensors

```
julia> A = ITensor(i,j)
```



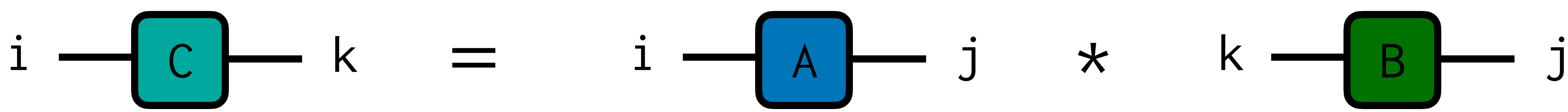

```
julia> B = ITensor(k,j)
```



How to contract matching indices?

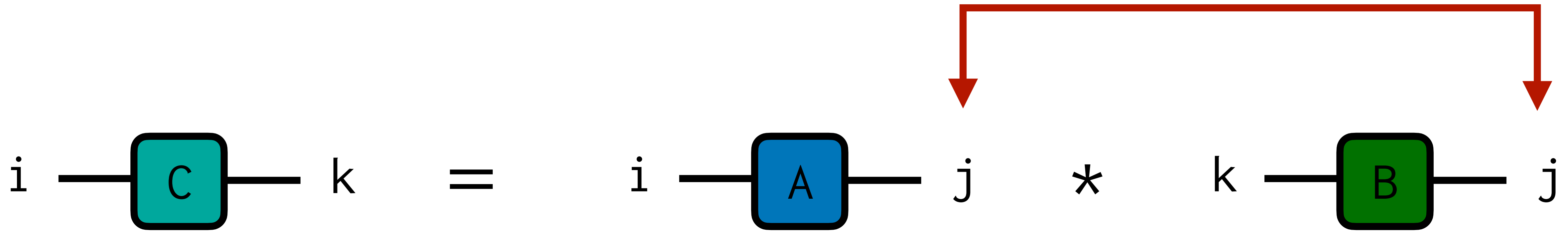
Contract ITensors with "*"

```
julia> C = A * B
```



Contract ITensors with "*"

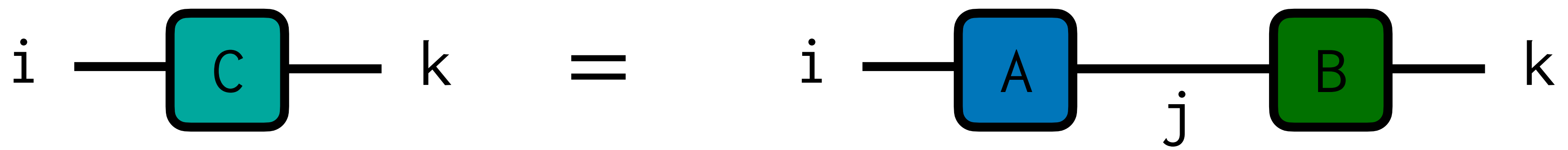
```
julia> C = A * B
```



Matching indices recognized

Contract ITensors with "*"

```
julia> C = A * B
```

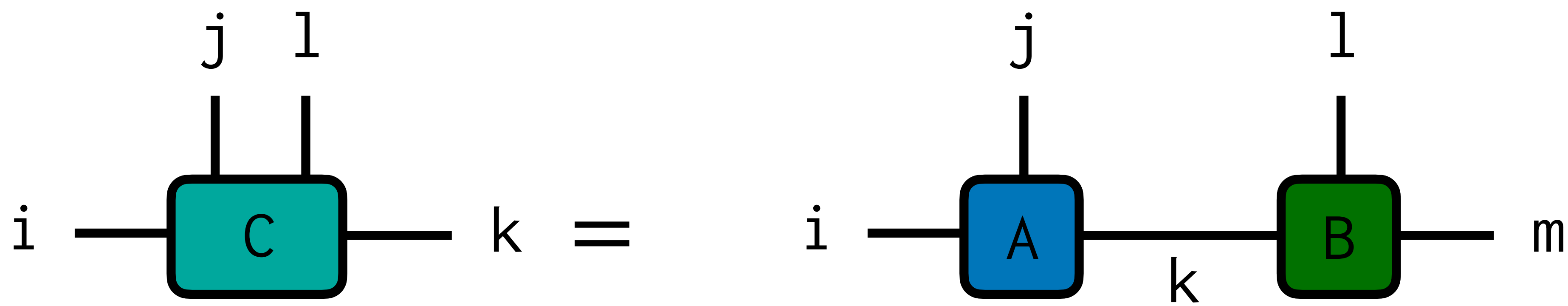


Matching indices recognized

Tensors permuted and contracted

General tensor contractions always

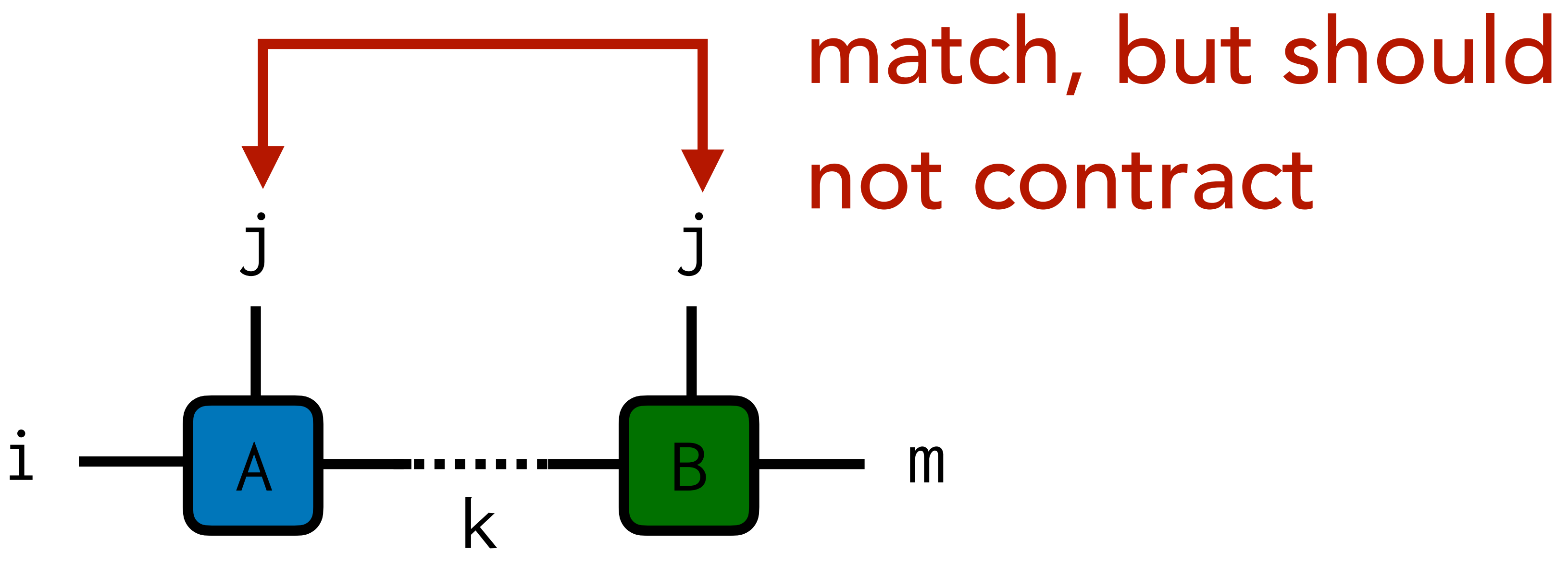
```
julia> C = A * B
```



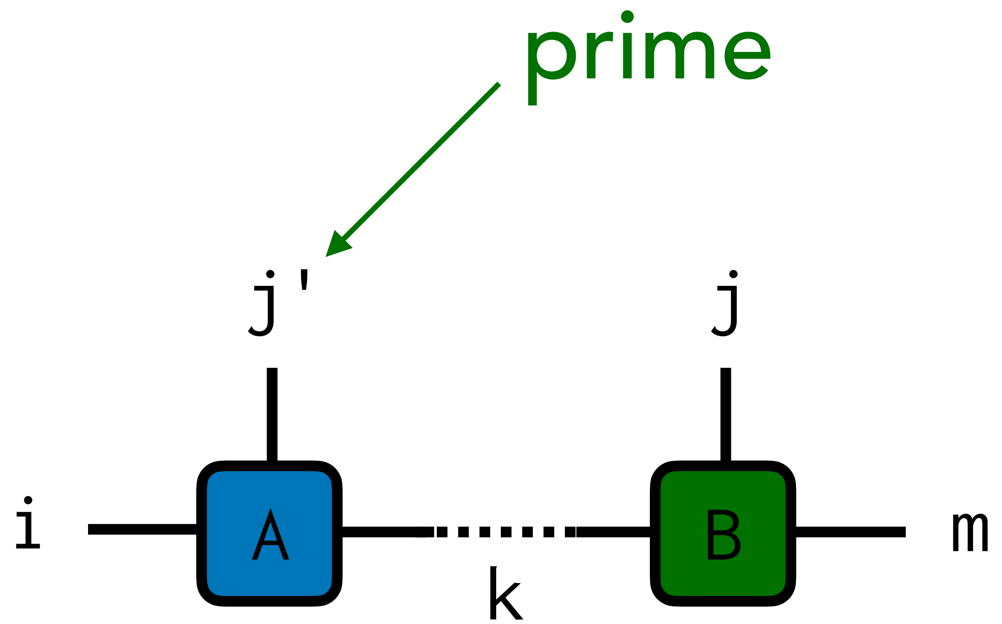
<u>Assuming</u> all matching indices should be contracted

Priming Indices

What if certain matching indices should stay uncontracted?

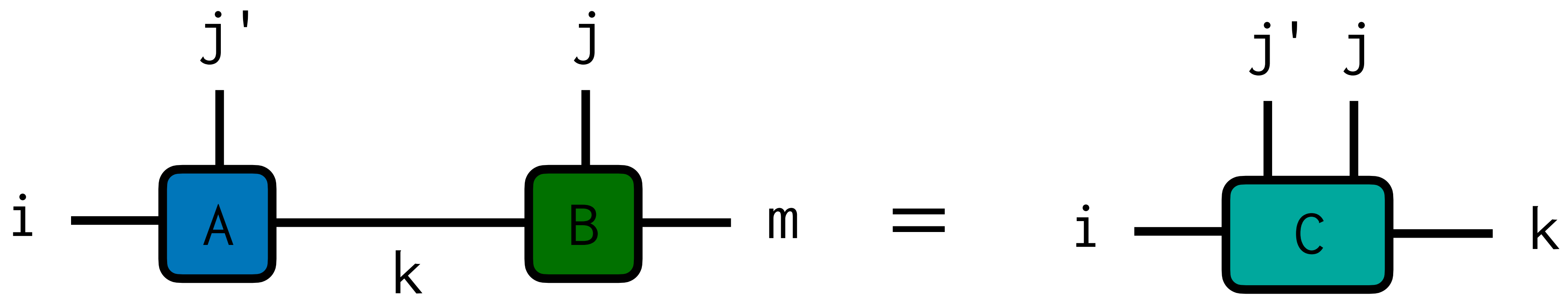


Prime an index to make it different



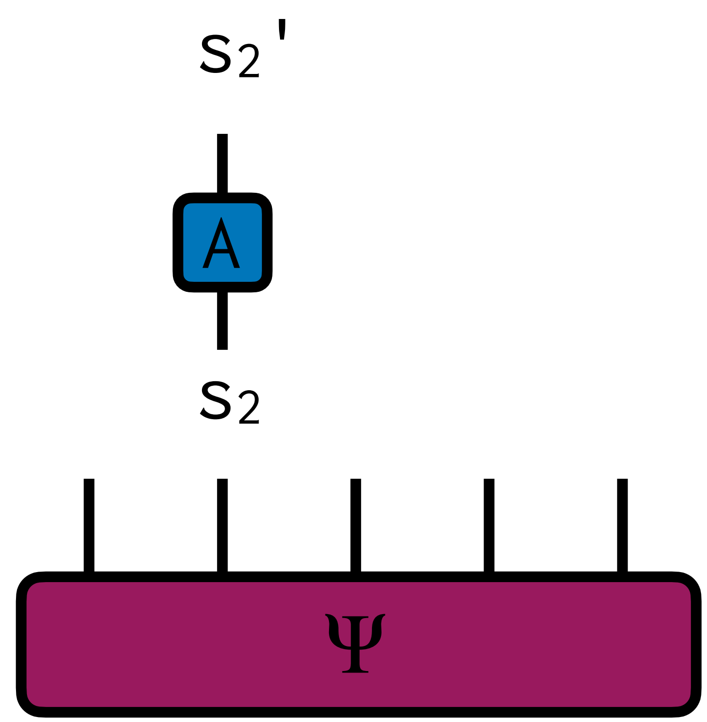

```
julia> prime(A,j)  # prime index j only
```

Contracting now gives desired result



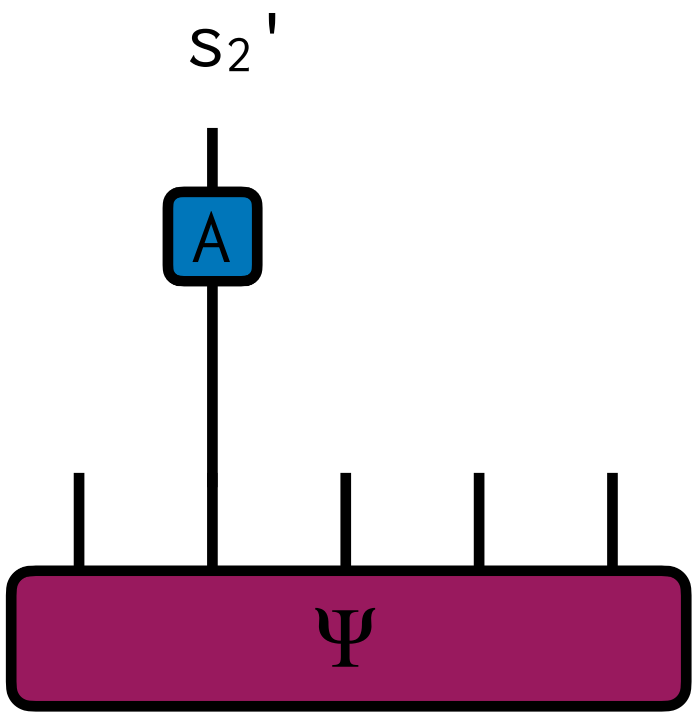

```
julia> Aprime = prime(A,j)  # prime index j only
julia> C = Aprime * B
```

Key use of priming is applying operators



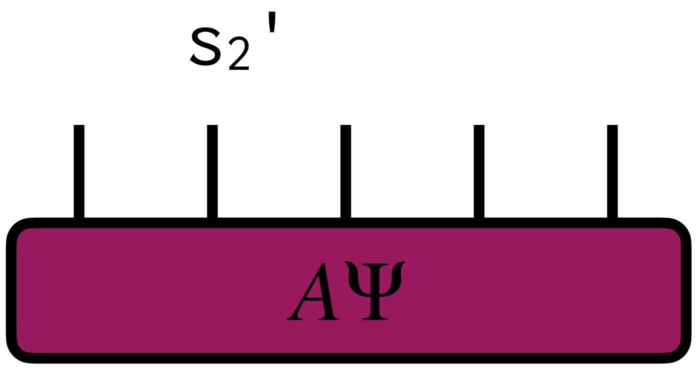

```
julia> A_psi = A * psi
```

Key use of priming is applying operators



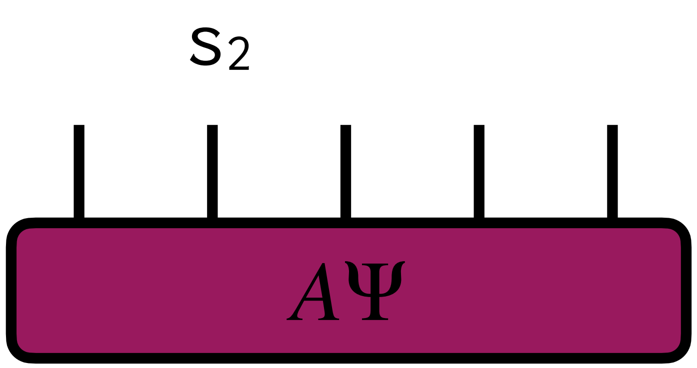

```
julia> A_psi = A * psi
```

Remove prime with `noprime` function



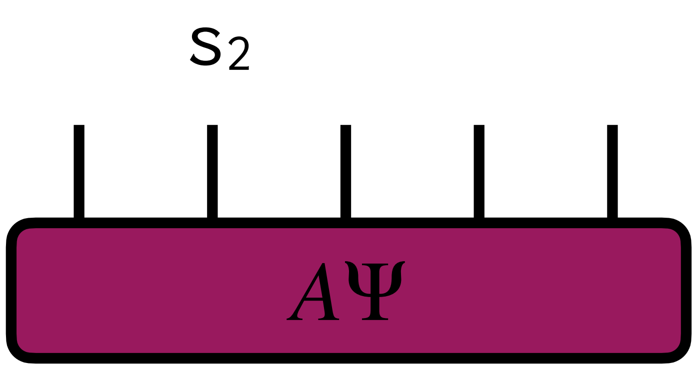

```
julia> A_psi = A * psi
julia> A_psi = noprime(A_psi)
```

Remove prime with `noprime` function



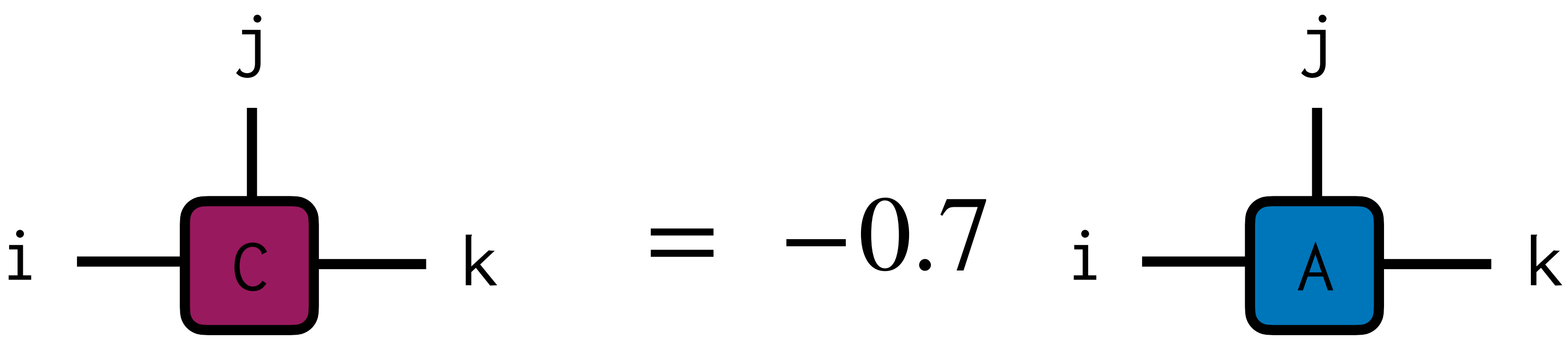

```
julia> A_psi = A * psi
julia> A_psi = noprime(A_psi)
```

Or use the shorthand `apply`

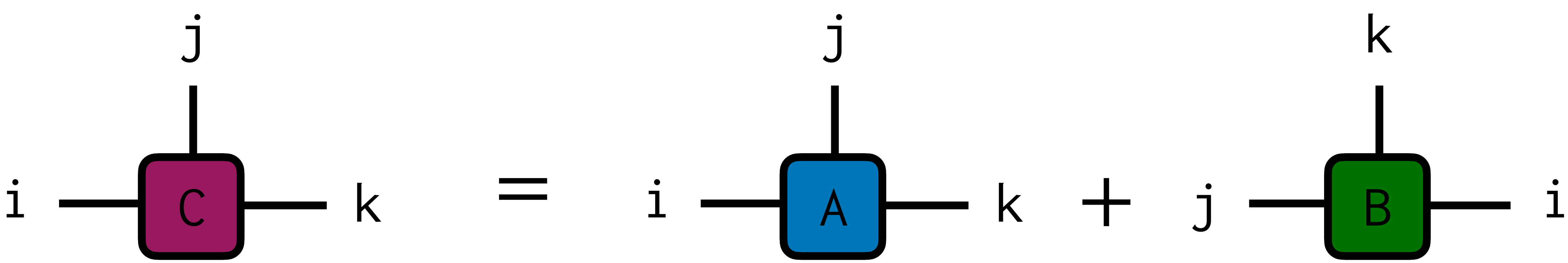



```
julia> A_psi = apply(A, psi)
```

Naturally, ITensors also support vector operators like multiplication by scalar and addition




```
julia> C = -0.7 * A
```

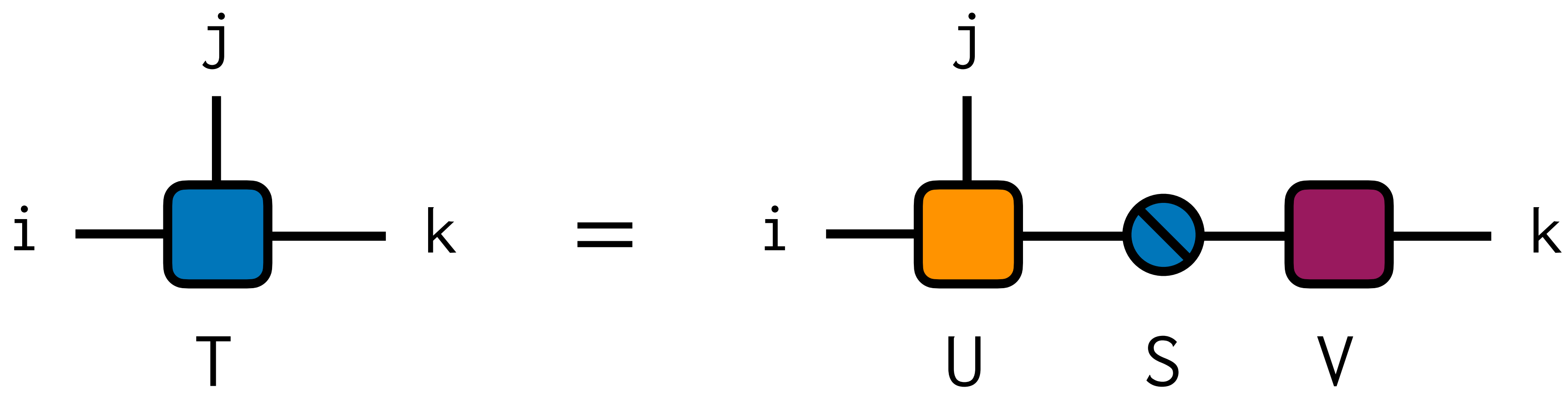



```
julia> C = A + B
```

Indices must match, but different order ok

ITensor Decompositions

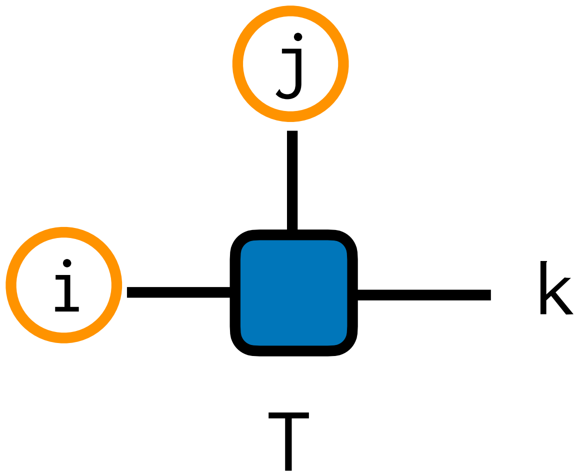
Recall we can SVD a tensor




```
julia> T ≈ U * S * V
true
```

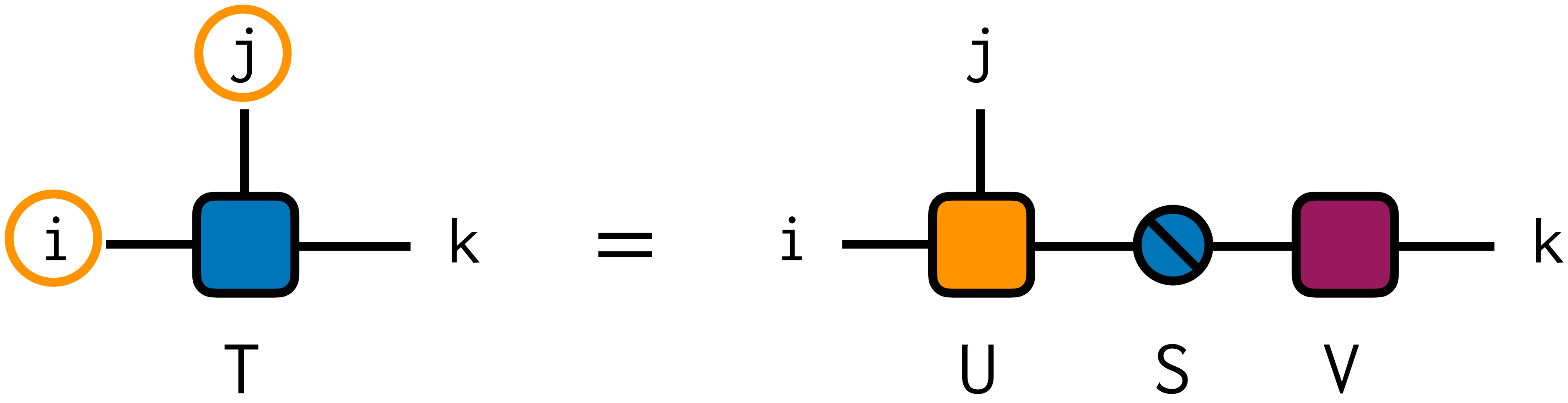
How to SVD an ITensor?

Select which indices go onto "U"



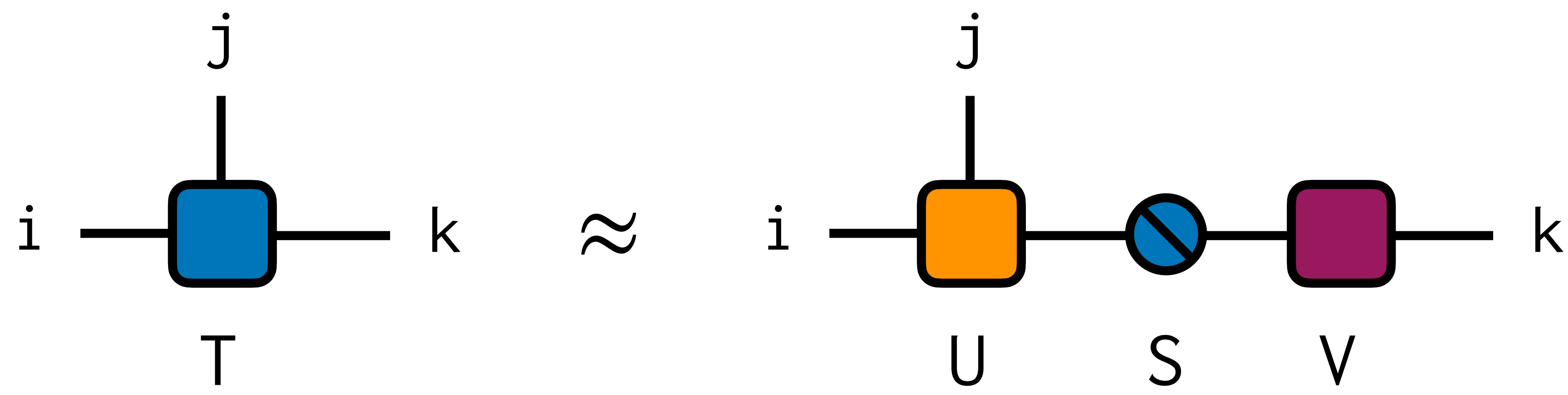

```
julia> u_inds = [i,j]
```

Call ITensor `svd` function




```
julia> u_inds = [i,j]
julia> U,S,V = svd(T, u_inds)
```

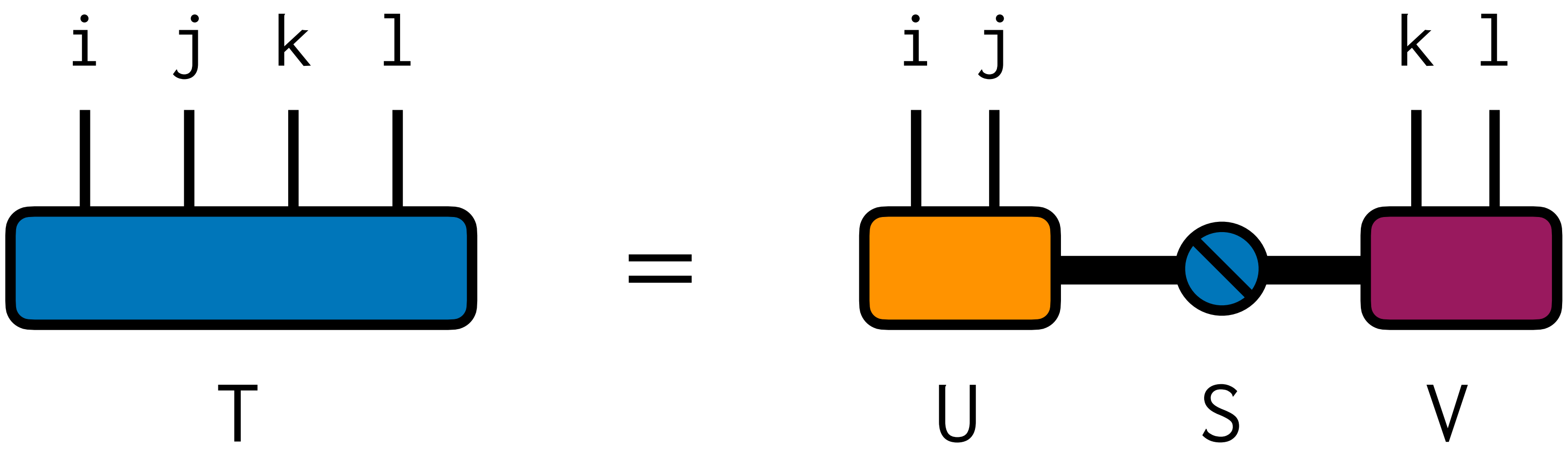
To compute truncated SVD, pass truncation parameters: maxdim, cutoff




```
julia> u_inds = [i,j]
julia> U,S,V = svd(T, u_inds; maxdim=10, cutoff=1E-4)
```

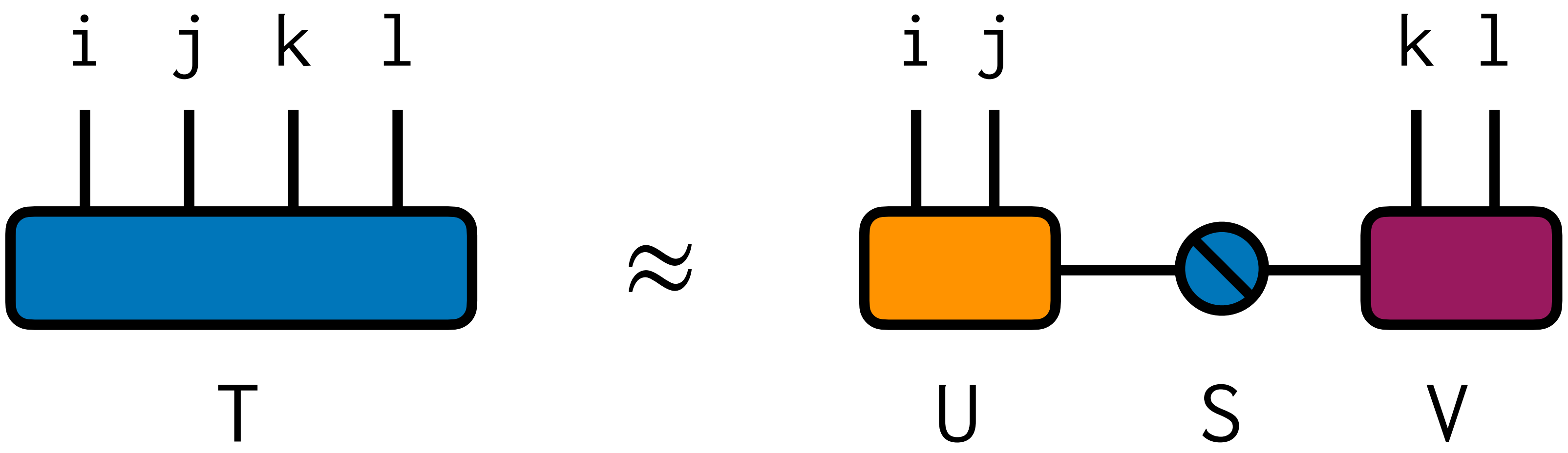
- maxdim: maximum dimension of internal bond
- cutoff ϵ: discard singular values λ_j such that $\sum_{j \in \text{discarded}} \lambda_j^2 \leq \epsilon$

Intuition about truncation - large outer spaces yet tensor has hidden "low rank" property




```
julia> u_inds = [i,j]
julia> U,S,V = svd(T, u_inds)
```

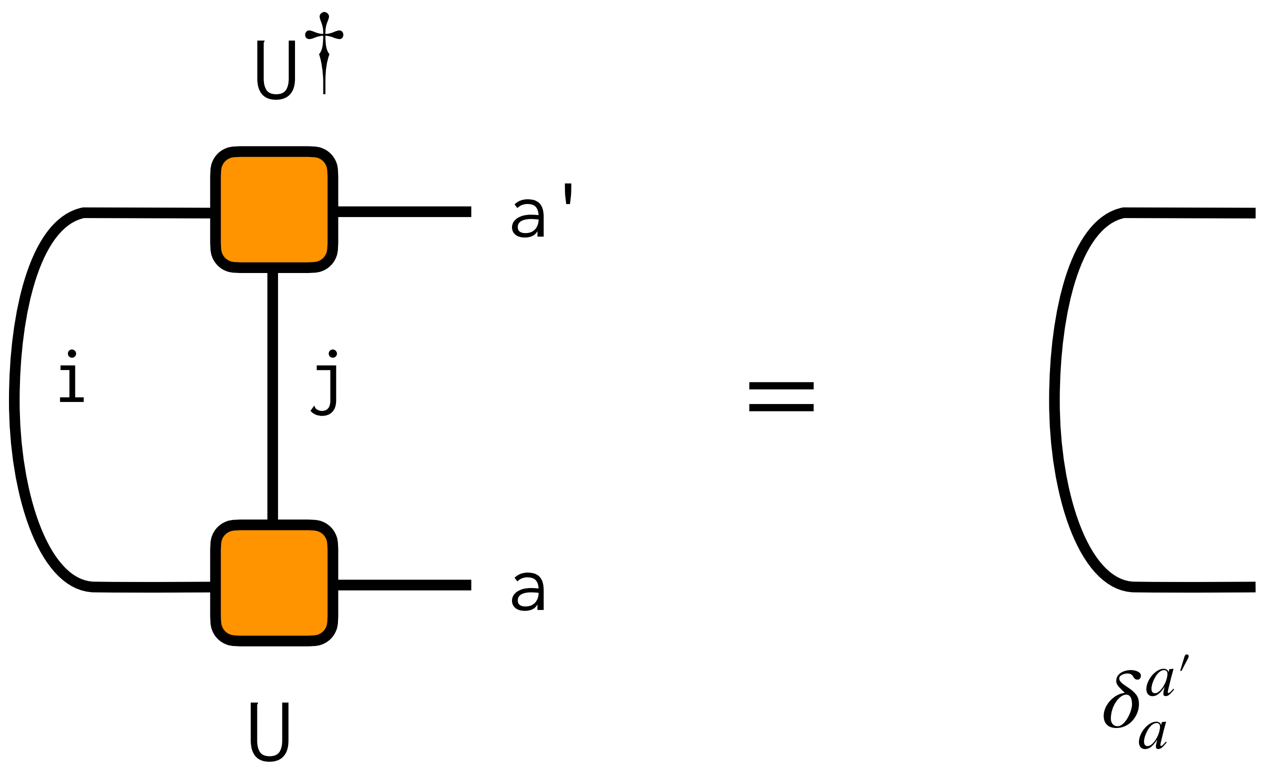
Intuition about truncation - large outer spaces yet tensor has hidden "low rank" property




```
julia> u_inds = [i,j]
julia> U,S,V = svd(T, u_inds; maxdim=10, cutoff=1E-4)
```

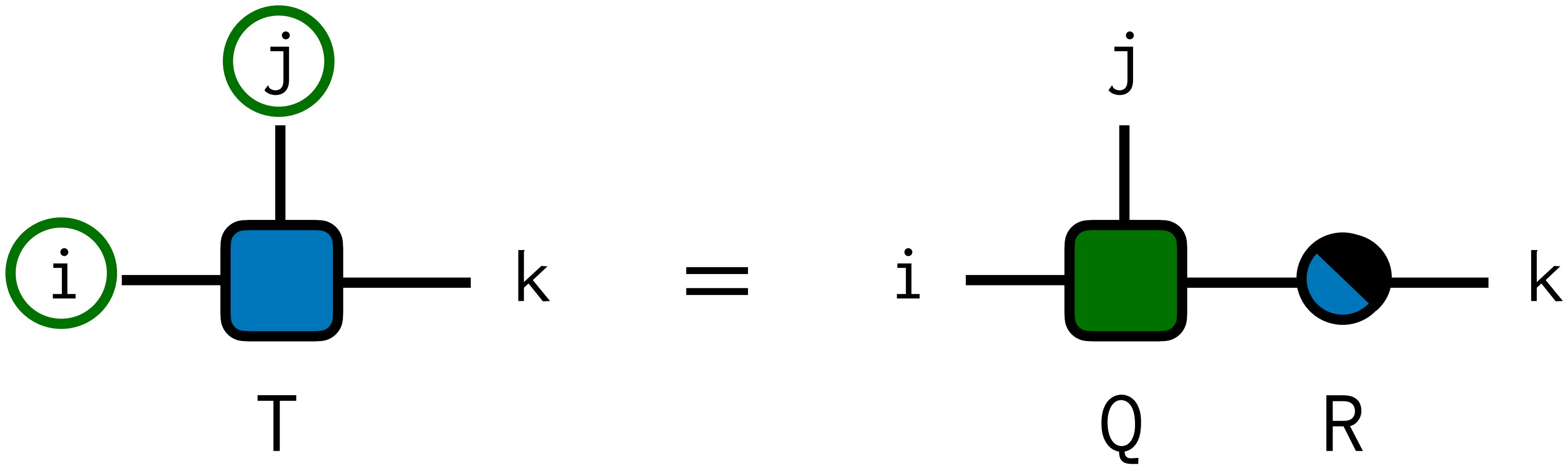
U and V have "isometric" property

One-sided 'unitary':



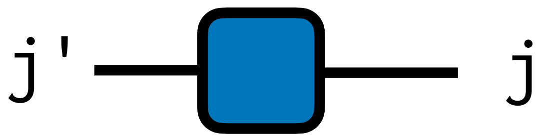
Other decompositions work similarly

QR, for example

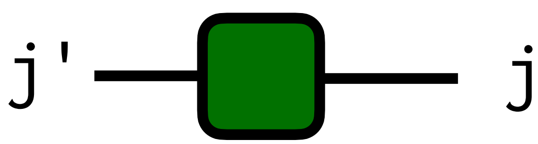



```
julia> q_inds = [i,j]
julia> Q,R = qr(T, q_inds)
julia> T ≈ Q * R
true
```

Also exponentiation of ITensors



A



$$= I + tA + \frac{t^2}{2!}A^2 + \dots$$

e^{tA}

```
julia> A = ITensor(j',j)
       ...
julia> exptA = exp(t * A)
```

ITensor Advanced Features
&
ITensor Ecosystem

GPU Support

Easily run on GPUs by 'adapting' data type to CuArray

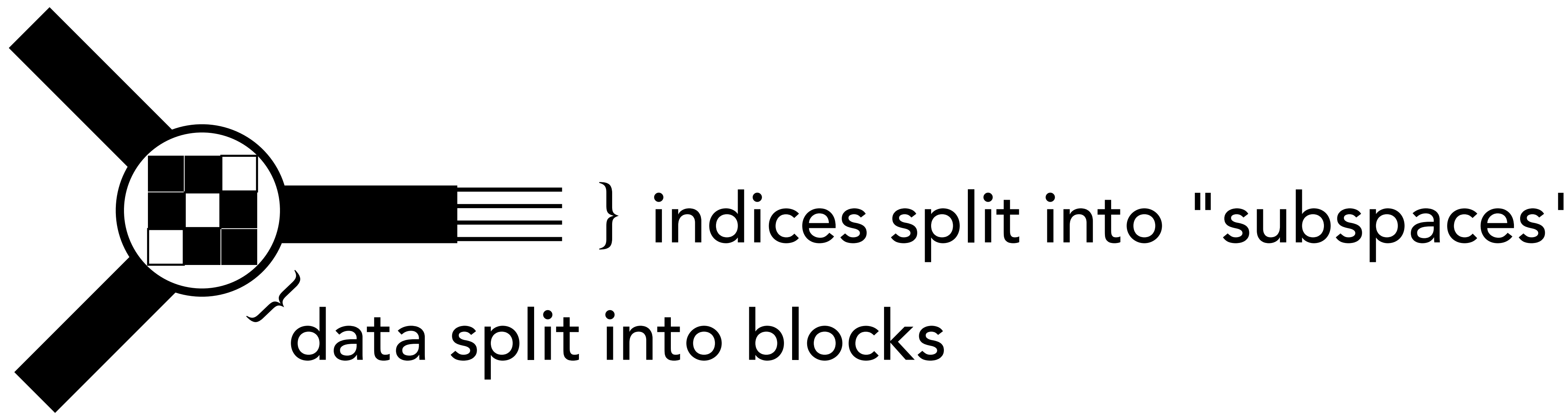
```
using ITensors, ITensorMPS
using Adapt, CUDA

N = 100
sites = siteinds("S=1/2", N)
terms = OpSum()
for j in 1:(N - 1)
  terms +=      "Sz", j, "Sz", j + 1
  terms += 1/2, "S+", j, "S-", j + 1
  terms += 1/2, "S-", j, "S+", j + 1
end
H = MPO(terms, sites)
psi0 = random_mps(sites; linkdims=10)


psi0 = adapt(CuArray, psi0)
H = adapt(CuArray, H)


nsweeps = 5
maxdim = [10, 20, 100, 100, 200]
cutoff = [1E-11]
energy, psi = dmrg(H, psi0; nsweeps, maxdim, cutoff)
```

Adapting the data type

Then use in DMRG, for example

Symmetric / Block-Sparse Tensors

Block sparse tensors in physics due to symmetries



ITensor: block information *inside indices* (index objects)

```
julia> i = Index([QN("Sz",0)=>2, QN("Sz",1)=>1, QN("Sz",-1)=>1])
(dim=4|id=444) <Out>
 1: QN("Sz",0) => 2
 2: QN("Sz",1) => 1
 3: QN("Sz",-1) => 1

julia> dim(i)
4
```

Summary

ITensor offers a novel interface with "intelligent" indices

Operations such as contraction and factorization use index system to ensure correctness

ITensor offers powerful features such as GPU support and symmetry (block-sparse tensors)

Up Next

After a break, we will get hands-on experience time-evolving MPS