Filling Sparse Dimensions

- So far, sparse dimensions have just been assumed as being pre-filled with the appropriate values
- There is nothing "magic" about this, if dusk & dawn wouldn't offer a means to fill sparse dimensions this would simply mean that it's the responsibility of the driver code to do so
- This choice may be appropriate if the sparse dimensions consist of quantities dependent on the mesh geometry only. So it can be precomputed once in an initialization step and then used throughout the simulation
    - This is due to the fact that the mesh doesn't deform in Eulerian simulations (e.g. climate applications)

- In ICON this is **NOT** the case, i.e. there are sparse fields dependent on the current velocities for example

→ dusk offers a concept to do so: `with sparse[CHAIN]:`

Filling Sparse Dimensions - Real World Example

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

Filling Sparse Dimensions - Real World Example

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

sparse index

Filling Sparse Dimensions - Real World Example

Neighbor Table for **Edge > Cell > Vertex**

```
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    vn_vert1(je,jk) = u_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v1 + &
                      v_vert(ividx(je,jb,1),jk,ivblk(je,jb,1)) * &
                      p_patch%edges%primal_normal_vert(je,jb,1)%v2

    vn_vert2(je,jk) = u_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v1 + &
                      v_vert(ividx(je,jb,2),jk,ivblk(je,jb,2)) * &
                      p_patch%edges%primal_normal_vert(je,jb,2)%v2

    vn_vert3(je,jk) = u_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v1 + &
                      v_vert(ividx(je,jb,3),jk,ivblk(je,jb,3)) * &
                      p_patch%edges%primal_normal_vert(je,jb,3)%v2

    vn_vert4(je,jk) = u_vert(ividx(je,jb,4),jk,ivblk(je,jb,4)) * &
                      !---SNIP----
```

Filling Sparse Dimensions - Real World Example - Dusk Version

```
@stencil
def ICON_laplacian_diamond(
    u_vert: Field[Vertex, K],
    v_vert: Field[Vertex, K],
    primal_normal_x: Field[Edge > Cell > Vertex],
    primal_normal_y: Field[Edge > Cell > Vertex],
    vn_vert: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn_vert = u_vert * primal_normal_x + v_vert * primal_normal_y
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (nIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
            vn(vIdx, linear_idx, k) +=
                u(vIdx,k)*nx(eIdx, linear_idx)
                v(vIdx,k)*ny(eIdx, linear_idx)
            linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (nIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx)
              v(vIdx,k)*ny(eIdx, linear_idx)
          linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx)
              v(vIdx,k)*ny(eIdx, linear_idx)
          linear_idx++
      }
  }
}
```

Filling Sparse Dimensions - Real World Example - Emitted Pseudocode

```
@stencil
def ICON_laplacian_diamond(
    u: Field[Vertex, K],
    v: Field[Vertex, K],
    nx: Field[Edge > Cell > Vertex],
    ny: Field[Edge > Cell > Vertex],
    vn: Field[Edge > Cell > Vertex, K]):
    with levels_upward:
        with sparse[Edge > Cell > Vertex]:
            vn =  u*nx + v*ny
```

```
parfor (k = 0; k < kmax; k++) {
  parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
      linear_idx = 0
      for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
          vn(eIdx, linear_idx, k) +=
              u(vIdx,k)*nx(eIdx, linear_idx, k)
              v(vIdx,k)*ny(eIdx, linear_idx, k)
          linear_idx++
      }
  }
}
```

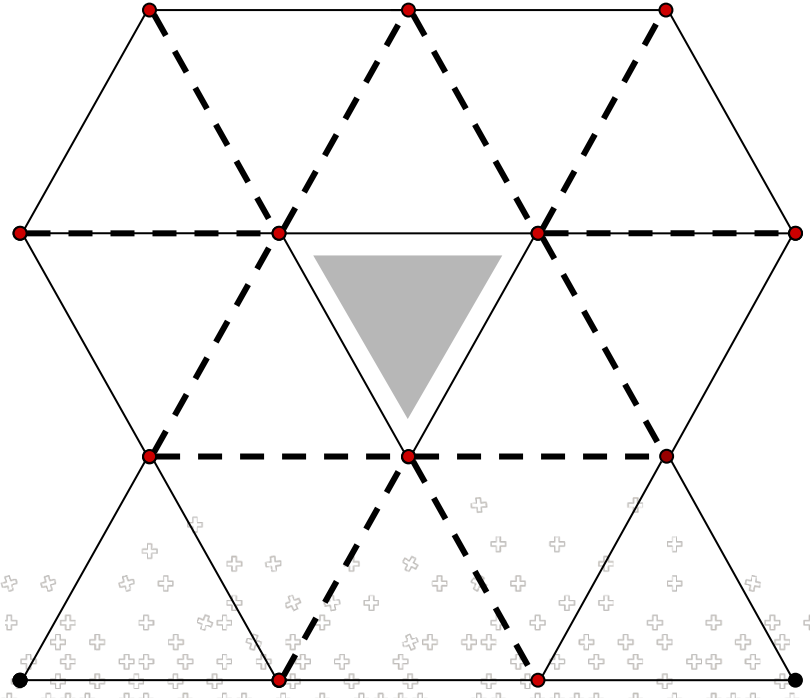
Filling Sparse Dimensions - Computing Interpolation Coefficients

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

- Coeffs depend on positions only, so this could be handled in driver code
- Still an interesting case to look at
- Works for arbitrary neighborhoods. Let's pick one we didn't consider so far: Cell -> Vertex -> Edge -> Vertex

Filling Sparse Dimensions - Computing Interpolation Coefficients

Cell -> Vertex -> **Edge** -> Vertex



Filling Sparse Dimensions - Computing Interpolation Coefficients

$$\langle f(\mathbf{p})\rangle_{\text{intp}} = \frac{1}{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)} \boxed{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)\cdot f(\mathbf{q}_i)}$$

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

$$W_h(\mathbf{p}-\mathbf{q}_i) = \frac{1}{h\pi^2} e^{\left(\frac{||\mathbf{p}-\mathbf{q}_i||^2}{h^2}\right)}$$

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

`with` statement encloses a *block* of statements
→can introduce intermediary results

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

such temporary fields need to be declared (for now)

`with` statement encloses a *block* of statements
→can introduce intermediary results

Filling Sparse Dimensions - Computing Interpolation Coefficients

```
@stencil
def kernelfuns(xn: Field[Vertex], yn: Field[Vertex],
               xc: Field[Cell], yc: Field[Cell],
               h: Field[Cell], pi: Field[Cell],
               wij: Field[Cell > Vertex > Edge > Vertex]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)
```

no support for globals / scalar arguments yet, need to waste a lot of memory

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p})\rangle_{\text{intp}} = \frac{1}{\sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)} \sum_{i\in\mathcal{N}(\mathbf{p})} W(\mathbf{p}-\mathbf{q}_i)\cdot f(\mathbf{q}_i)$$

```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex],
        wij: Field[Cell > Vertex > Edge > Vertex], Wn: Field[Cell]):
    qij: Field[Cell > Vertex > Edge > Vertex]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex]):
    wij: Field[Cell > Vertex > Edge > Vertex]
    qij: Field[Cell > Vertex > Edge > Vertex]
    Wn: Field[Cell]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$

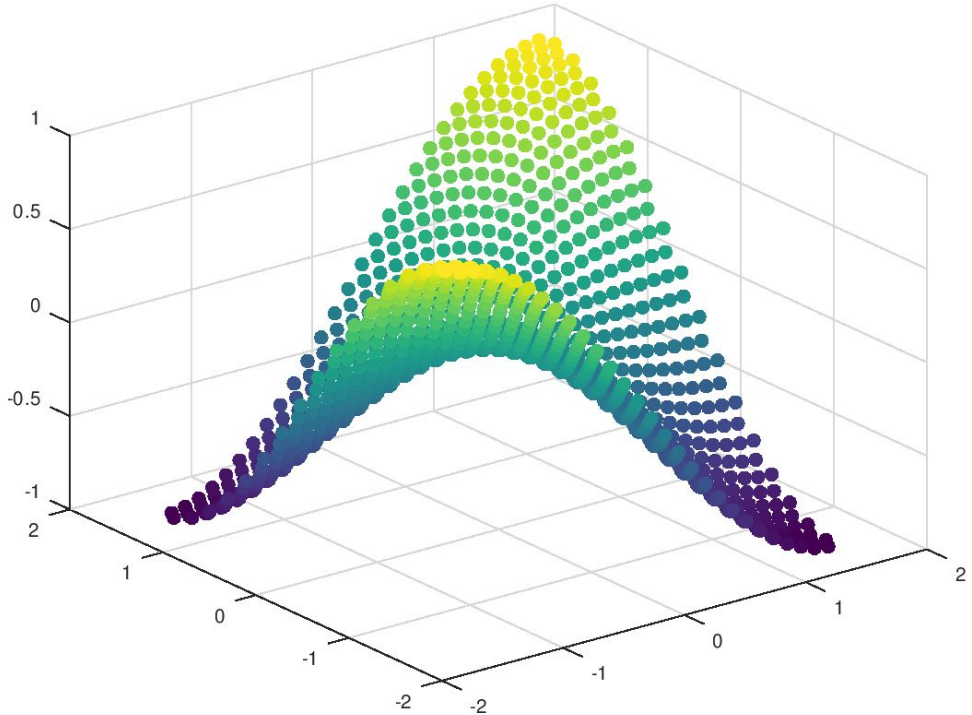
```
@stencil
def sph(xn: Field[Vertex], yn: Field[Vertex], xc: Field[Cell], yc: Field[Cell],
        fn: Field[Vertex], f_intp: Field[Cell], h: Field[Vertex], pi: Field[Vertex]):
    wij: Field[Cell > Vertex > Edge > Vertex]
    qij: Field[Cell > Vertex > Edge > Vertex]
    Wn: Field[Cell]
    with levels_upward:
        with sparse[Cell > Vertex > Edge > Vertex]:
            qij = sqrt((xc-xn)*(xc-xn) + (yc-yn)*(yc-yn))/h
            wij = 1./(pi*h*h)*exp(-qij*qij)

        Wn = sum_over(Cell > Vertex > Edge > Vertex, wij)
        f_intp = sum_over(Cell > Vertex > Edge > Vertex, 1/Wn*wij*fn)
```

Quick Excursion: Writing the complete interpolation

```
for (int kIter = klo; kIter < khi; kIter++) {
  if (kIter >= kSize + 0) { return; }
  ::dawn::float_type lhs_157 = (::dawn::float_type)0;
  for (int nbhIter = 0; nbhIter < C_V_E_V_SIZE; nbhIter++) {
    int nbhIdx = cvevTable[pidx * C_V_E_V_SIZE + nbhIter];
    if (nbhIdx == DEVICE_MISSING_VALUE) { continue;}
    lhs_157 += wij[nbhIter * NumCells + pidx];
  }
  ::dawn::float_type __local_Wn_1_115 = lhs_157;
  ::dawn::float_type lhs_161 = (::dawn::float_type)0;
  for (int nbhIter = 0; nbhIter < C_V_E_V_SIZE; nbhIter++) {
    int nbhIdx = cvevTable[pidx * C_V_E_V_SIZE + nbhIter];
    if (nbhIdx == DEVICE_MISSING_VALUE) { continue; }
    lhs_161 +=
        ((((int)1 / __local_Wn_1_115) * wij[nbhIter * NumCells + pidx]) *
         fn[nbhIdx]);
  }
  f_intp[pidx] = lhs_161;
```

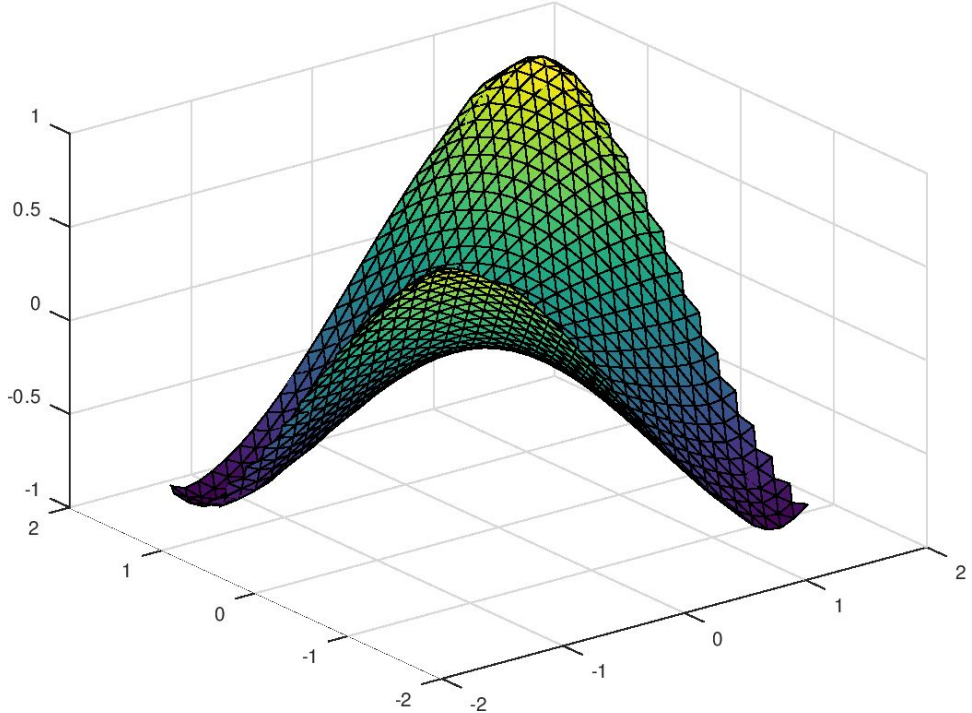
Quick Excursion: Writing the complete interpolation

$$\langle f(\mathbf{p}) \rangle_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot \boxed{f(\mathbf{q}_i)}$$



Quick Excursion: Writing the complete interpolation

$$\boxed{\langle f(\mathbf{p}) \rangle}_{\text{intp}} = \frac{1}{\sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i)} \sum_{i \in \mathcal{N}(\mathbf{p})} W(\mathbf{p} - \mathbf{q}_i) \cdot f(\mathbf{q}_i)$$



Sparse Dimensions and their Type

- We have seen quite a few examples of sparse dimensions now
- Let's take a step back and think about their (location) type
- An ordinary field has a single (location) type

```
eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex]
```

- A sparse field has a chain of locations as (location) type

```
eceField: Field[Edge > Cell > Edge], ecvField: Field[Edge > Cell > Vertex]
```

Sparse Dimensions and their Type

- A statement involving ordinary fields only (and is consistent in it's type) is generated into a loop over it's location type

```
@stencil
def dense(
    a: Field[Edge], b: Field[Edge],
    c: Field[Cell], d: Field[Cell]):
    with levels_downward:
        a = b
        c = d
```



```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
     a(eIdx) = b(eIdx)
for (cIdx = 0; cIdx < mesh.num_cells(); cIdx++)
     c(cIdx) = d(cIdx)
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = b*sparseF
```

?

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
  linear_idx = 0
  for (nIdx : mesh.nbh(eIdx, {Edge, Cell})) {
    a(eIdx) = b(eIdx)*sparseF(eIdx,linear_idx)
    linear_idx++
  }
}
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = b*sparseF
```

?

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
  linear_idx = 0
  for (nIdx : mesh.nbh(eIdx, {Edge, Cell})) {
    a(eIdx) += b(eIdx)*sparseF(eIdx,linear_idx)
    linear_idx++
  }
}
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A with sparse statement

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A **reduction** expressions
  - A `with sparse` statement

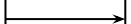
```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge],
    sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = sum_over(Edge>Cell, b*sparseF)
```

```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
  linear_idx = 0
  for (cIdx : mesh.nbh(eIdx, {Edge, Cell})) {
     a(eIdx) += b(eIdx)*sparseF(eIdx,linear_idx)
     linear_idx++
  }
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement

```
@stencil
def sparse(
   a: Field[Edge], b: Field[Edge],
   sparseF: Field[Edge>Cell]):
   with levels_downward:
     with sparse[Edge>Cell]:
       sparseF = b*a
```



```
for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
  linear_idx = 0
  for (cIdx : mesh.nbh(eIdx, {Edge, Cell})) {
     sparseF(eIdx,linear_idx) = a(eIdx) * b(eIdx)
     linear_idx++
  }
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the sparse dimension stated in the first argument of the reduction / the parameter of the `with sparse` statement.

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the **sparse dimension** stated in the **first argument of the reduction** / the parameter of the `with sparse` statement.

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge], sparseF: Field[Edge>Cell]):
    with levels_downward:
        a = sum_over(Edge>Cell, b*sparseF)
```

Sparse Dimensions and their Type

- Now what about statements involving sparse fields?
- Since the semantic is unclear we restricted to accesses to sparse field to be either contained in:
  - A reduction expressions
  - A `with sparse` statement
- The (location) type consistency rules for the sparse dimension itself are quite simple:
  - The sparse dimensions involved in a reduction or a sparse fill concept need to match the **sparse dimension** stated in the first argument of the reduction / the **parameter of the** `with sparse` statement.

```
@stencil
def sparse(
    a: Field[Edge], b: Field[Edge], sparseF: Field[Edge>Cell]):
    with levels_downward:
      with sparse[Edge>Cell]:
        sparseF = b*a
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * ?)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * ?)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * eField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * vField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * cField)
```

ILLEGAL CODE

Sparse Dimensions and their Type

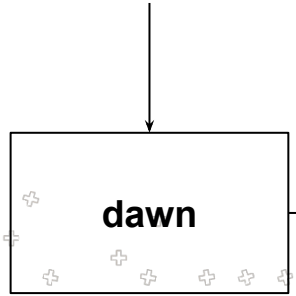
- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF * eField *
                                                   vField)
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



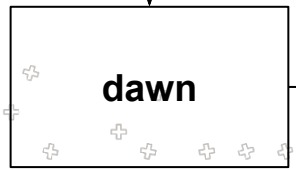
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx, k)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



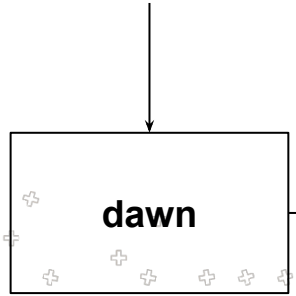
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
   with levels_downward:
       eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



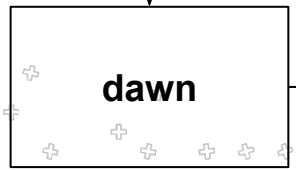
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                    sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], nField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex], eOut: Field[Edge]):
    with levels_downward:
        eOut = sum_over(Edge > Cell > Vertex, sparseF * eField * vField)
```



MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        eOut(eIdx) += eField(eidx)*vField(vIdx)
                      sparseF(eIdx, linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex]
):
    with levels_downward:
        with sparse[Edge > Cell > Vertex]:
            sparseF = eField*vField
```

Sparse Dimensions and their Type

- What about the dense fields involved in a reduction using a neighbor chain?
  →Dense Fields need to match either start or end type of reductions location chain!
- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
    eField: Field[Edge], cField: Field[Cell], vField: Field[Vertex],
    sparseF: Field[Edge > Cell > Vertex]
):
    with levels_downward:
        with sparse[Edge > Cell > Vertex]:
            sparseF = cField
```

ILLEGAL CODE

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```

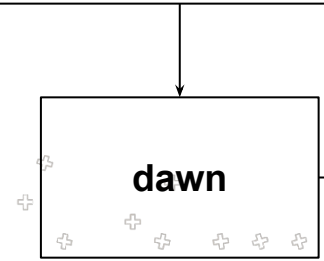

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



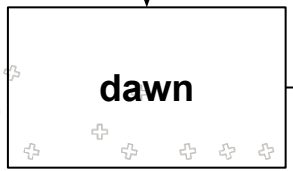

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



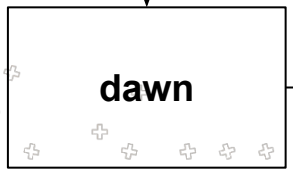
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- What about the dense fields involved in a `with sparse` statement?
  →Dense Fields need to match either start or end type of `with sparse` location chain!

```
@stencil
def mixed_fields(
   eField: Field[Edge], vField: Field[Vertex],
   sparseF: Field[Edge > Cell > Vertex]
):
   with levels_downward:
       with sparse[Edge > Cell > Vertex]:
           sparseF = eField*vField
```



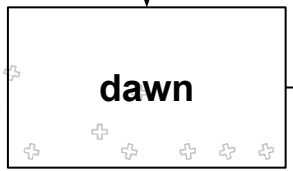

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (vIdx : mesh.nbh(eIdx,{Edge, Cell, Vertex})) {
        sparseF(eIdx, linearIdx) =
            eField(eidx)*vField(vIdx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```



MeteoSwiss

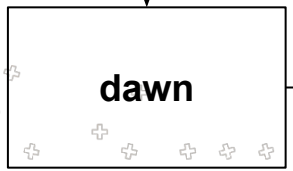
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!



MeteoSwiss

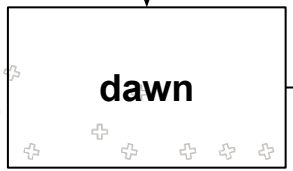
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!



MeteoSwiss

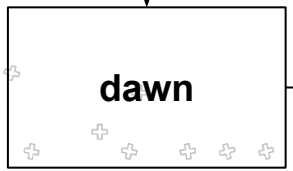
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify



MeteoSwiss

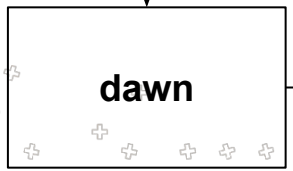
```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(?)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF = eField[Edge]
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify



MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(edgeIdx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Let's look at one more example!

```
@stencil
def mixed_fields(
    eField: Field[Edge],
    sparseF: Field[Edge > Cell > Edge]
):
    with levels_downward:
        with sparse[Edge > Cell > Edge]:
            sparseF =
                eField[Edge>Cell>Edge]
```

→Situation is ambiguous!
→User could expect `eField` to be read at either `eIdx` or `eIdxInner`!
→dawn recognizes such situations and asks the user to clarify

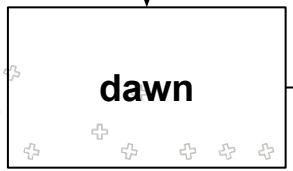
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        sparseF(eIdx, linearIdx) = eField(eIdxInner)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Same principle for reductions applies

```
@stencil
def mixed_fields(
   eIn: Field[Edge],
   eOut: Field[Edge],
   sparseF: Field[Edge > Cell > Edge]
):
   with levels_downward:
    eOut = sum_over(Edge > Cell > Edge,
                    sparseF*eIn[Edge])
```



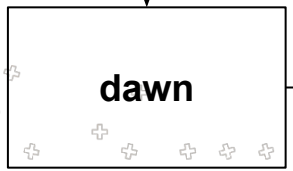
MeteoSwiss

```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        eOut(eIdx)+= eIn(eIdx)*sparseF(eIdx,linear_idx)
        linear_idx++
    }
}
```

Sparse Dimensions and their Type

- Same principle for reductions applies

```
@stencil
def mixed_fields(
   eIn: Field[Edge],
   eOut: Field[Edge],
   sparseF: Field[Edge > Cell > Edge]
):
   with levels_downward:
    eOut = sum_over(Edge > Cell > Edge,
       sparseF*eIn[Edge > Cell > Edge])
```




```
parfor (eIdx = 0; eIdx < mesh.num_edges(); eIdx++) {
    linear_idx = 0
    for (eIdxInner : mesh.nbh(eIdx,{Edge, Cell, Edge})) {
        eOut(eIdx)+=
            eIn(eIdxInner)*sparseF(eIdx,linear_idx)
        linear_idx++
    }
}
```

MeteoSwiss

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these **horizontal offsets**

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Vertex], eFieldOut: Field[Edge]):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Vertex, sparseF[Edge > Cell > Vertex] * eField[Edge])
```

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambiguous cases, user is *required* to state horizontal offsets

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField[Edge])
```

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField)
```

DuskSyntaxError: Field 'eField' requires a horizontal index inside of ambiguous neighbor iteration!

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets
- Horizontal offsets are *never allowed* on the left hand side of an assignment

Sparse Dimensions and their Type - The Horizontal Offset

Ambiguous Cases:

- In non-ambiguous cases, user *may* state these horizontal offsets
- In ambigous cases, user is *required* to state horizontal offsets
- Horizontal offsets are *never allowed* on the left hand side of an assignment

```
@stencil
def mixed_fields(
    eField: Field[Edge], sparseF: Field[Edge > Cell > Edge],
    eFieldOut: Field[Edge]
):
    with levels_downward:
        eFieldOut[Edge] = sum_over(Edge > Cell > Edge, sparseF[Edge > Cell > Edge] * eField[Edge])
```

DuskSyntaxError: Invalid horizontal index for field 'eFieldOut' outside of neighbor iteration!

Nested Reductions

- Meaningful semantics can be assigned to nested reductions
- Early stage, not very well tested
- No code generation for cuda backend
  - No (apparent) use case in ICON dycore

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Outer Reduction

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Inner Reduction

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Inner Reduction
→ is executed |Edge > Cell| = 2 times

Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Start of inner chain needs to match end of outer chain

Nested Reductions

Example

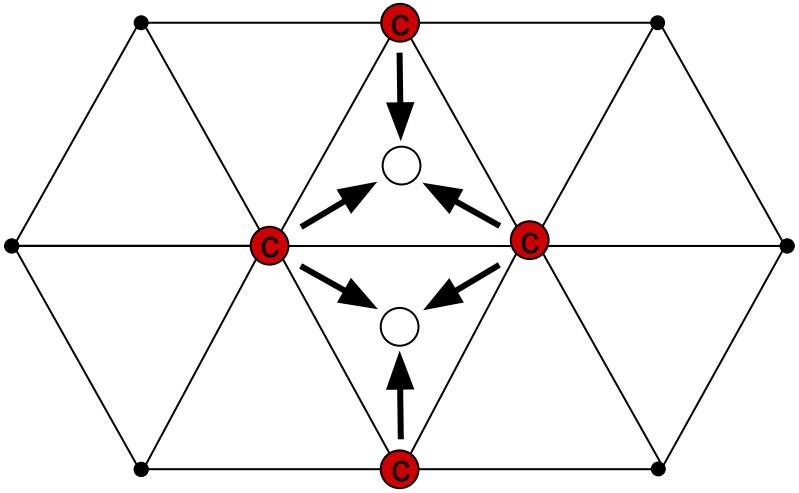
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

Start of inner chain is return type of inner reduction
→ needs to form a (location) type consistent expression

Nested Reductions

Example

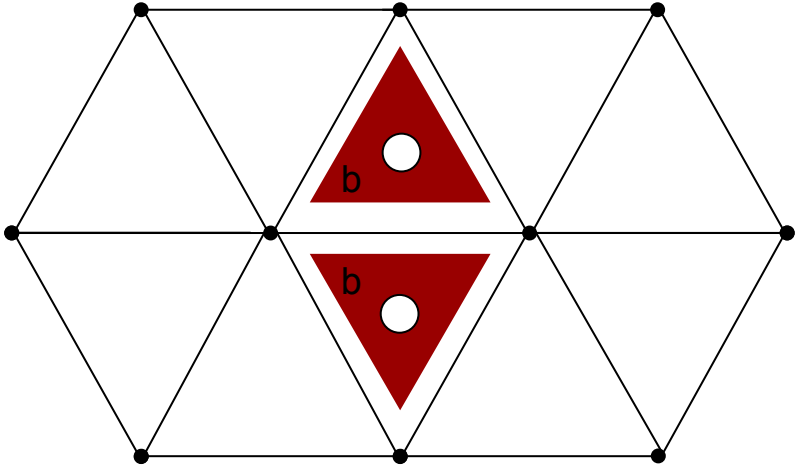
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

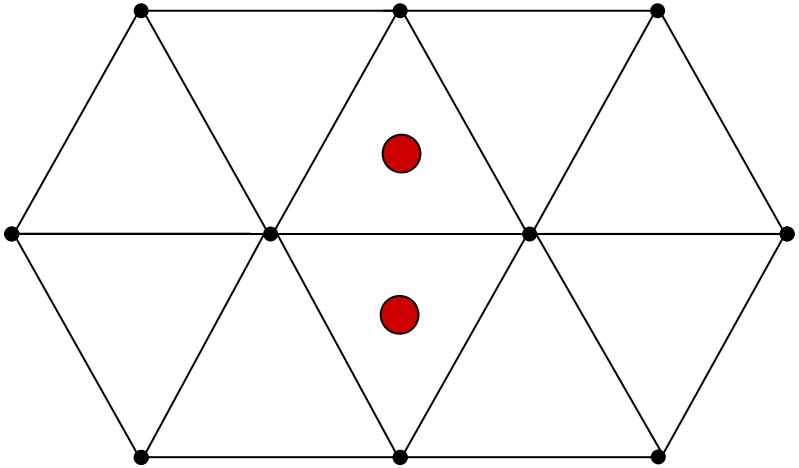
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

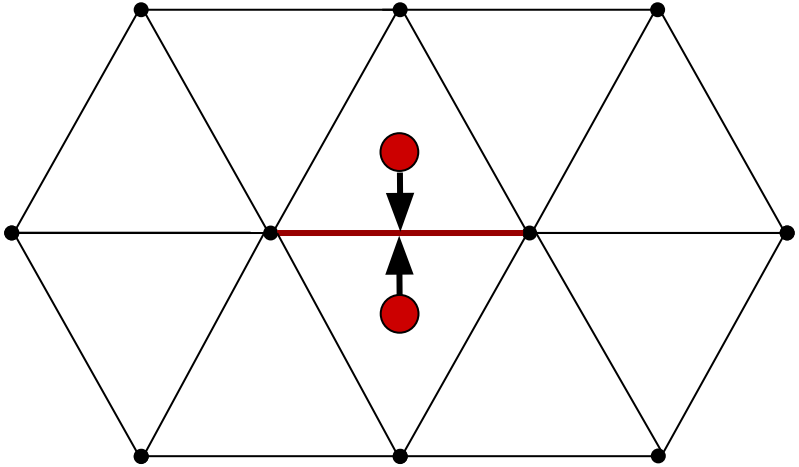
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

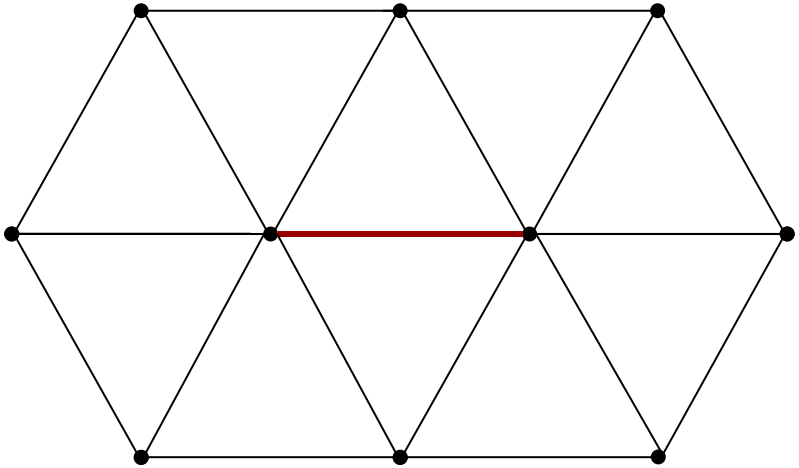
```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions

Example

```
@stencil
def nested(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                b*reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



Nested Reductions - Comparison to reduction with neighbor chain

Consider the following two examples:

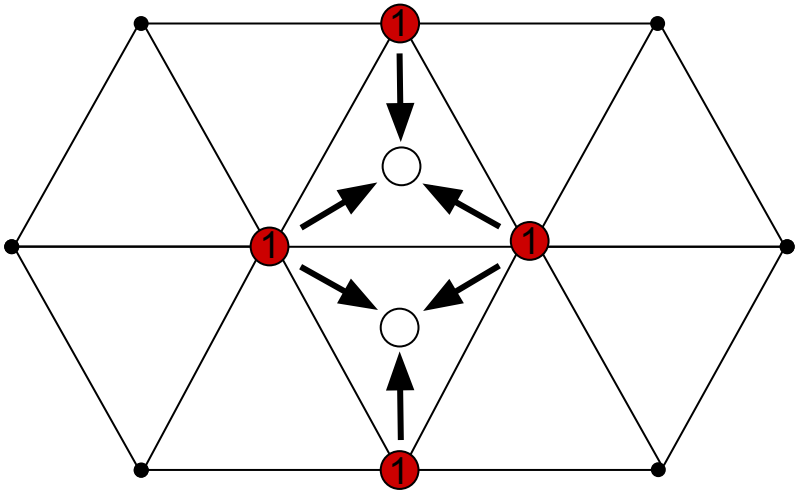
- are they different? equal?
- let's assume c(:) = 1. a(:) = ?

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```

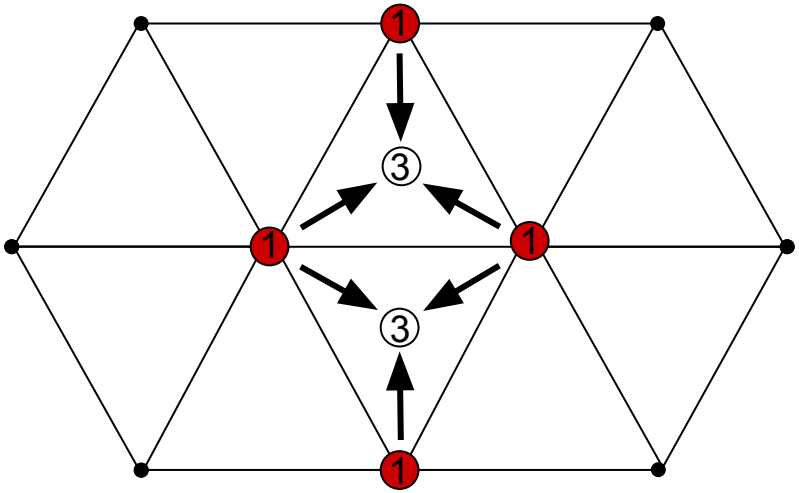
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



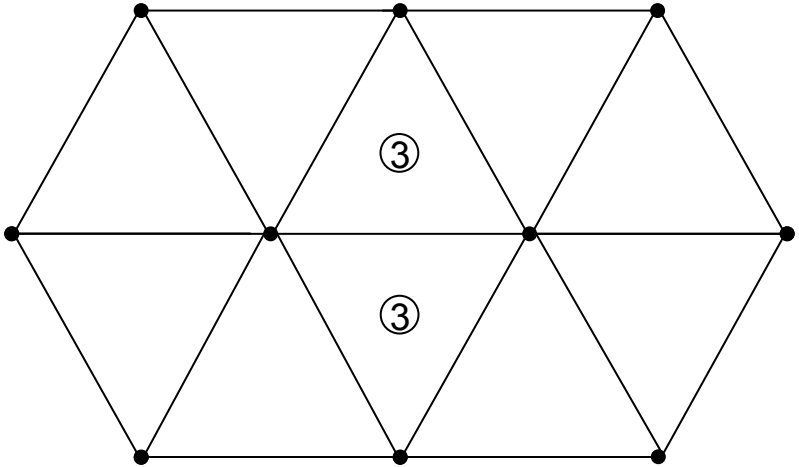
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



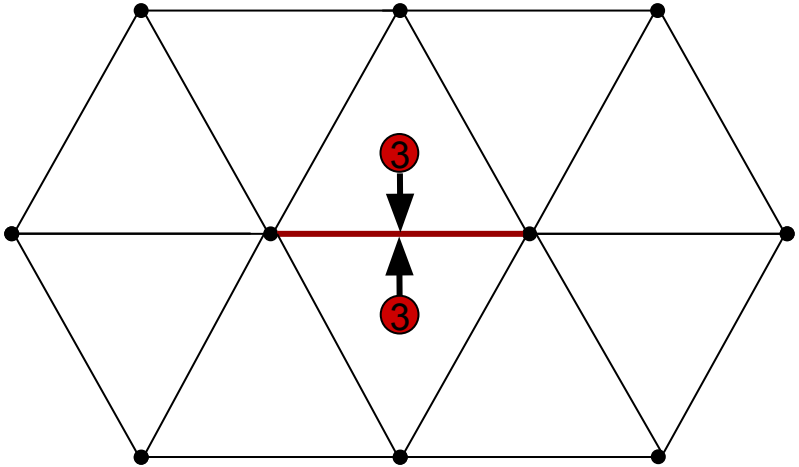
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



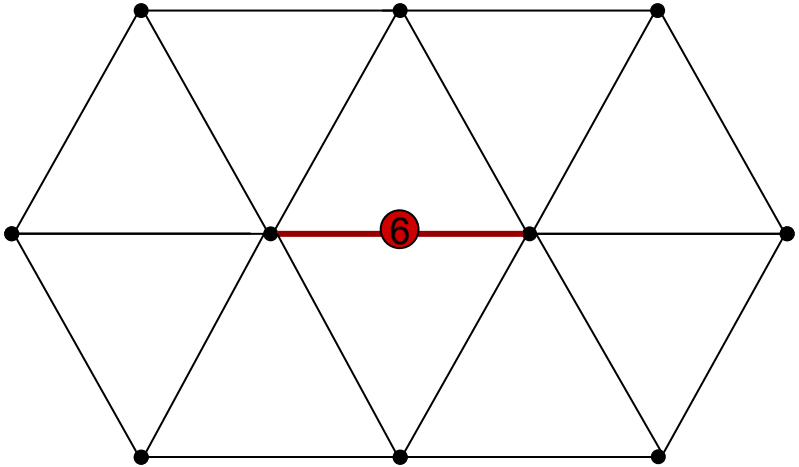
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



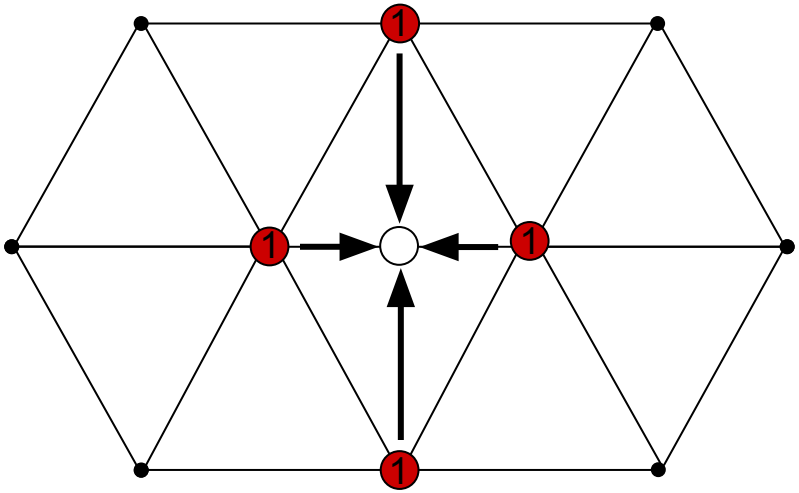
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def nested(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell,
                reduce_over(Cell > Vertex, c, sum, init=0.0),
            sum, init=0.0)
```



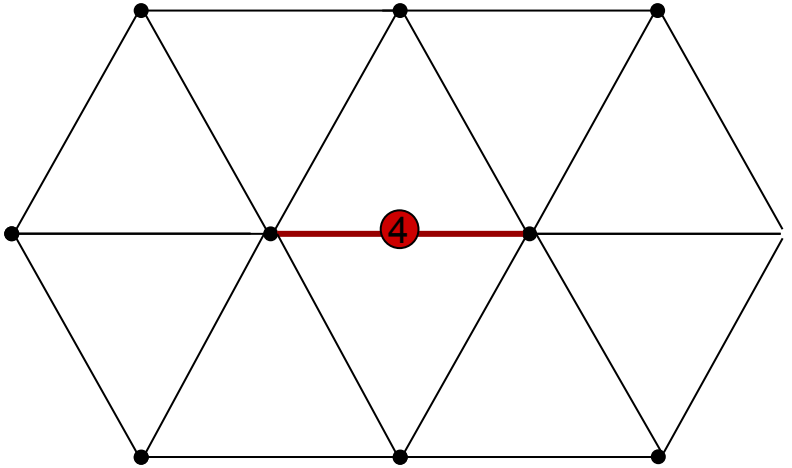
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```

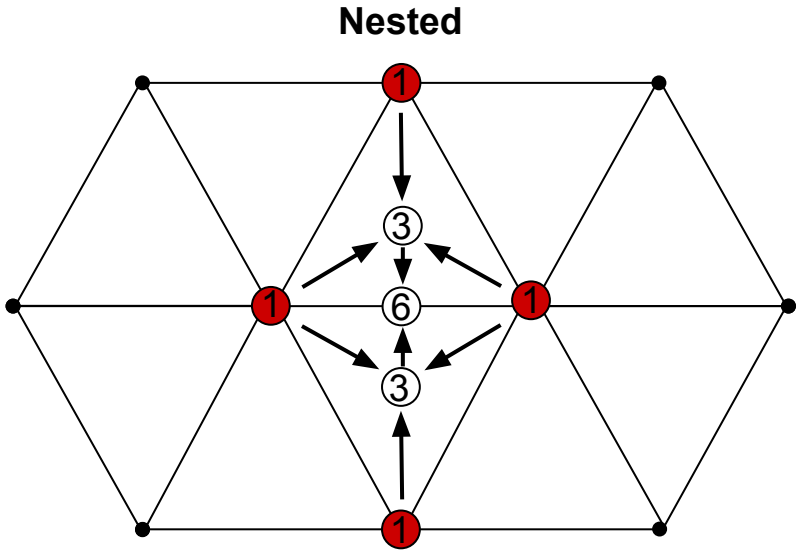


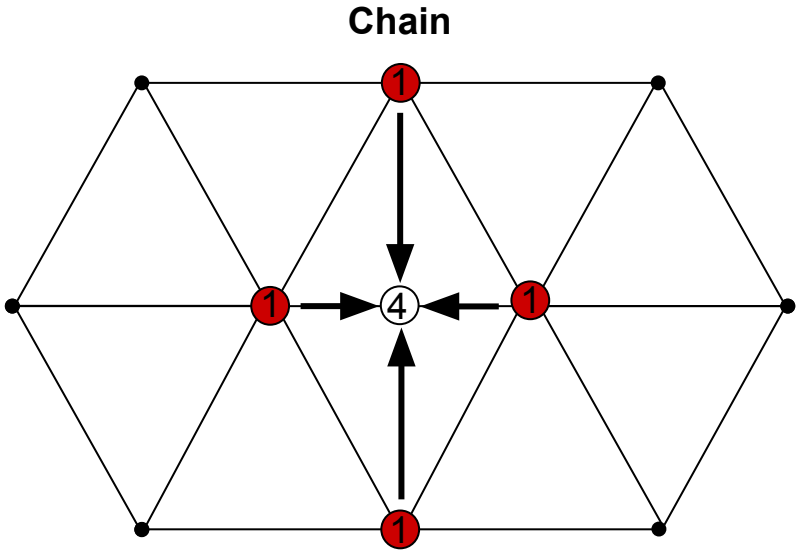
Nested Reductions - Comparison to reduction with neighbor chain

```
@stencil
def chained(
    a: Field[Edge], c: Field[Vertex]
):
    with levels_downward:
        a = reduce_over(Edge > Cell > Vertex,
                c, sum, init=0.0)
```



Nested Reductions - Comparison to reduction with neighbor chain





Nested Reductions - Pushing the Envelope

Concepts are compatible / composable. Nothing stops you from...

- … nesting three levels (n levels) deep
- ... mixing nested with chained reductions
- … adding sparse fields

or even doing all of the above

Nested Reductions - Pushing the Envelope

Concepts are compatible / composable. Nothing stops you from...

- … nesting three levels (n levels) deep
- ... mixing nested with chained reductions
- … adding sparse fields

or even doing all of the above

```
@stencil
def wat(
    a: Field[Edge], b: Field[Cell], c: Field[Vertex],
    sparseOuter: Field[Edge>Cell], sparseInner: Field[Vertex > Cell > Edge > Vertex]
):
    with levels_downward:
        a = sum_over(Edge > Cell,
                b*sparseOuter*sum_over(Cell > Vertex,
                    sum_over(Vertex > Cell > Edge > Vertex,
                        sparseInner[Vertex > Cell > Edge > Vertex]*c[Vertex > Cell > Edge > Vertex])))
```

Nested Reductions - Stencil Inlining

Stencil Inlining

```
@stencil
def reference(
field_a: Field[Vertex, K],
field_c: Field[Edge, K]
):
    field_b: Field[Cell, K]
    with levels_upward:
        field_b = sum_over(Cell > Vertex,
                           field_a)
        field_c = sum_over(Edge > Cell,
                           field_b)
```

inline

```
@stencil
def inlined(
field_a: Field[Vertex, K],
field_c: Field[Edge, K]
):
    with levels_upward:
        field_c = sum_over(Edge > Cell,
                           sum_over(Cell > Vertex, field_a))
```

Nested Reductions - Stencil Inlining

Stencil Inlining

- can be done automatically in an optimization pass
- has performance implications
- more on that later

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset. A quick reminder:

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset. A quick reminder:

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K]):
    with levels_upward as k:
        outF = inF[k+1]
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension

```
@stencil
def der_k(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[1:-1] as k:
        inF_k = (inF[k+1] - inF[k-1])/h[k]
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal

```
@stencil
def der_k(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[1:-1] as k:
        inF_k = (inF[k+1] - inF[k-1])/h[k]

@stencil
def f_k_high_order(inF: Field[Edge, K], inF_k: Field[Edge, K], h: Field[Edge, K]):
    with levels_upward[0:-2] as k:
        inF_k = (inF[k+2] - 4*inF[k+1] + inF[k])/(2*h[k])
```

Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal
- dusk & dawn is more flexible than that, the offset can also be read from a field instead:

```
@stencil
def indirect_read(inF: Field[Edge, K], outF: Field[Edge, K], indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection]
```

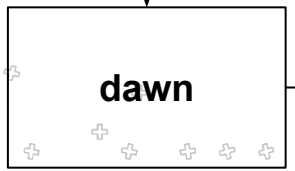
Vertical Offsets & Indirections

- We have seen in the very beginning how the `with levels_downward:` and `with levels_upward:` concepts can be used to read a field vertically offset.
- This is, for example, useful to compute Finite Difference Stencils along the vertical dimension
- Note that in all of these examples, the offset is a compile time literal
- dusk & dawn is more flexible than that, the offset can also be read from a field instead:
  - we can still combine this with a compile time literal offset

```
@stencil
def indirect_read(inF: Field[Edge, K], outF: Field[Edge, K], indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection+1]
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[k]
```

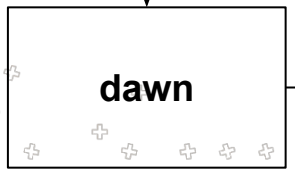


MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,k)
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection]
```

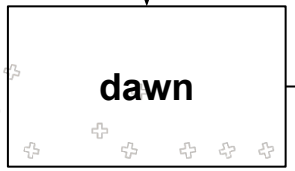


MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,indirection(eIdx,k))
```

Vertical Offsets & Indirections

```
@stencil
def offset_read(inF: Field[Edge, K], outF: Field[Edge, K],
                indirection: IndexField[Edge, K]):
  with levels_upward as k:
      outF = inF[indirection+1]
```



MeteoSwiss

```
for (k = 0; k < kmax; k++)
   for (eIdx = 0; eIdx < mesh.num_edges(); eIdx++)
      outF(eIdx,k) = inF(eIdx,indirection(eIdx,k)+1)
```

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K]):
  with levels_upward as k:
      outF[k+1] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K], indir: IndexField[Edge, K]):
  with levels_upward as k:
      outF[indir] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*

```
@stencil
def offset_write(inF: Field[Edge, K], outF: Field[Edge, K], indir: IndexField[Edge, K]):
  with levels_upward as k:
      indir = indir + 5
      outF[indir] = inF
```

ILLEGAL CODE

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*
  - Need to either prepare them in driver code, or split stencil and switch type

```
@stencil
def prepare(v_vertical: Field[Edge, K], indir: Field[Edge, K], dt: Field[Edge, K]):
  with levels_upward as k:
      indir = -v_vertical * dt

@stencil
def use(indir: IndexField[Edge, K], ...):
  with levels_upward as k:
      ...
```

Vertical Offsets & Indirections

Some constraints:

- The vertical offset can not be used on the left hand side to do offset *writes*
- Index fields can not be used to do *indirected writes*
- Index fields are *read only*
  - Need to either prepare them in driver code, or split stencil and switch type

```
@stencil
def prepare(v_vertical: Field[Edge, K], indir: Field[Edge, K], dt: Field[Edge, K]):
  with levels_upward as k:
      indir = -v_vertical * dt

@stencil
def use(indir: IndexField[Edge, K], ...):
  with levels_upward as k:
      ...
```

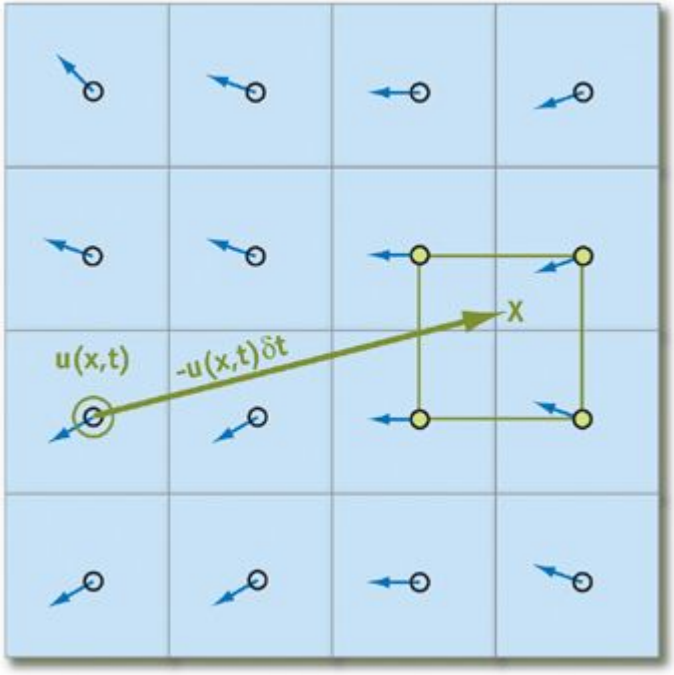
Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical

```
@stencil
def compute_btraj(v_vertical: Field[Cell, K], dt: Field[Cell,K], btraj: Field[Cell, K]):
  with levels_upward as k:
      btraj = -floor(v_vertical/dt)
```

Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical



taken from GPU Gems Chpt 38

Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical

```
@stencil
def advectT(T: Field[Cell,K], v_vertical: Field[Cell, K],
            dt: Field[Cell,K], btraj: IndexField[Cell, K],
            levels: Field[Cell, K], Tnew: Field[Cell,K]):
  # temperatures at corners of departure region
  T_depart_hi: Field[Cell, K]
  T_depart_lo: Field[Cell, K]
  # height coordinate corners of departure region
  h_depart_hi: Field[Cell, K]
  h_depart_lo: Field[Cell, K]
  # height coordinate at departure _point_
  h: Field[Cell, K]
  # lerp'ed Temp at departure point
  T_lerp: Field[Cell, K]
  with levels_upward as k:
      ...
```

Vertical Offsets & Indirections

Typical Use Case: Semi-Lagrangian advection in the Vertical

```
...
with levels_upward as k:
    T_depart_hi = T[btraj+1]
    T_depart_lo = T[btraj]
    h_depart_hi = levels[btraj+1]
    h_depart_lo = levels[btraj]
    h = levels[k] - v_vertical*dt
    T_lerp = T_depart_lo +
               (h-h_depart_lo)*(T_depart_hi - T_depart_lo)/(h_depart_hi - h_depart_lo)
    Tnew = T_lerp
```