CSE462/562: Database Systems (Fall 24)

Lecture 9: Query Execution Models

9/24/2024

University at Buffalo
Department of Computer Science
and Engineering
School of Engineering and Applied Sciences

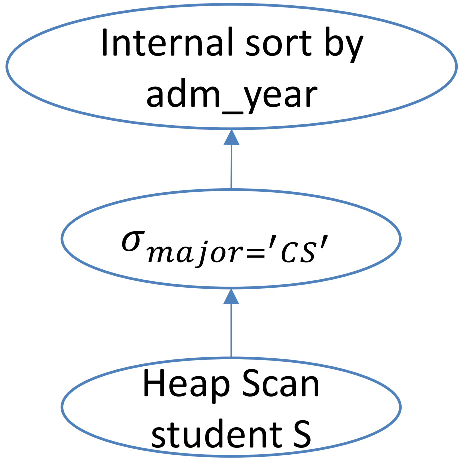
Last updated: 9/10/2024 9:35 AM

Query execution models

- Several models for implementing the operators
  - Volcano model (aka iterator model)
    - most traditional and widely used one
    - pull-based execution
  - Materialization model
  - Vectorization model

- Running example

```
SELECT * FROM student
WHERE major='CS' ORDER BY adm_year;
```



Volcano model

- Operators implemented as subclasses of some `iterator` interface similar to below

```
struct iterator {
  void init();
  Record next();
  void close();
  void rewind();
  Iterator *inputs[];
};
```

- *Encapsulation*
  - Edges are encoded as inputs (aka child iterators)
  - Each operator implementation maintains its own internal state in its subclass
  - Generally, any operator can be input to any other operators
- *Evaluation strategy: pull-based execution*
  - Call `next()` repeatedly on the root
  - Iterators recursively call `next()` on the inputs
    - Can be pipelining or materializing, depending on the operators
- Note: the iterator tree sometimes is a separate homomorphic tree to the physical plan
  - Allows caching of physical plan (read-only)
  - A new iterator tree for storing mutable execution state per query

Example: heap scan

```
struct heap_scan_iterator: public iterator {
    heap_scan_iterator(relation R) {   // leaf level, no input in heap scan
        table = create a Table object over R;
    }
    void init() {
       iter = create and initialize an iterator over t;   // initializing internal states
    }
    Record next() {
       if (iter.next()) {
           return the record in iter;
       }
       return an invalid record;
    }
    void close() {
       close the iterator and the table;
    }
    void rewind() {
       close and recreate a iterator in iter;
    }
   // internal state of a heap scan
    Table *table;
    Table::Iterator iter;
};
```

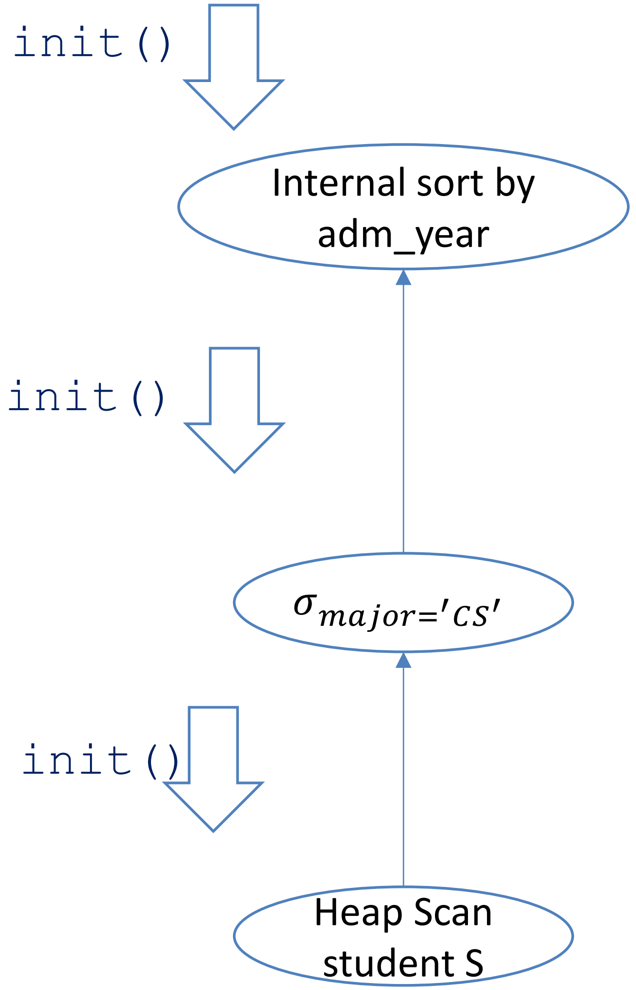
Example: selection *σ* (streaming)

```
struct selection_iterator: public iterator {
    selection_iterator(iterator *c, BooleanExpression *e): {
        set input[0] = c;   // selection has one input node
        set pred = e;
    }
    void init() {
        input[0]->init();   // iterator implementation must recursively initialize the inputs
    }
    Record next() {
        while (r = input[0]->next()) {   // call next on the input iterator to get the next record for selection
            if (pred evaluates to true on record r) { return r; }  // only return when pred is true
        }
        return an invalid record;
    }
    void close() {
        input[0]->close();
    }
    void rewind() {
        input[0]->rewind();
    }
   // internal state of a selection. note that no record is ever stored in the iterator
    BooleanExpression *pred;
};
```

Example: internal sort (blocking)

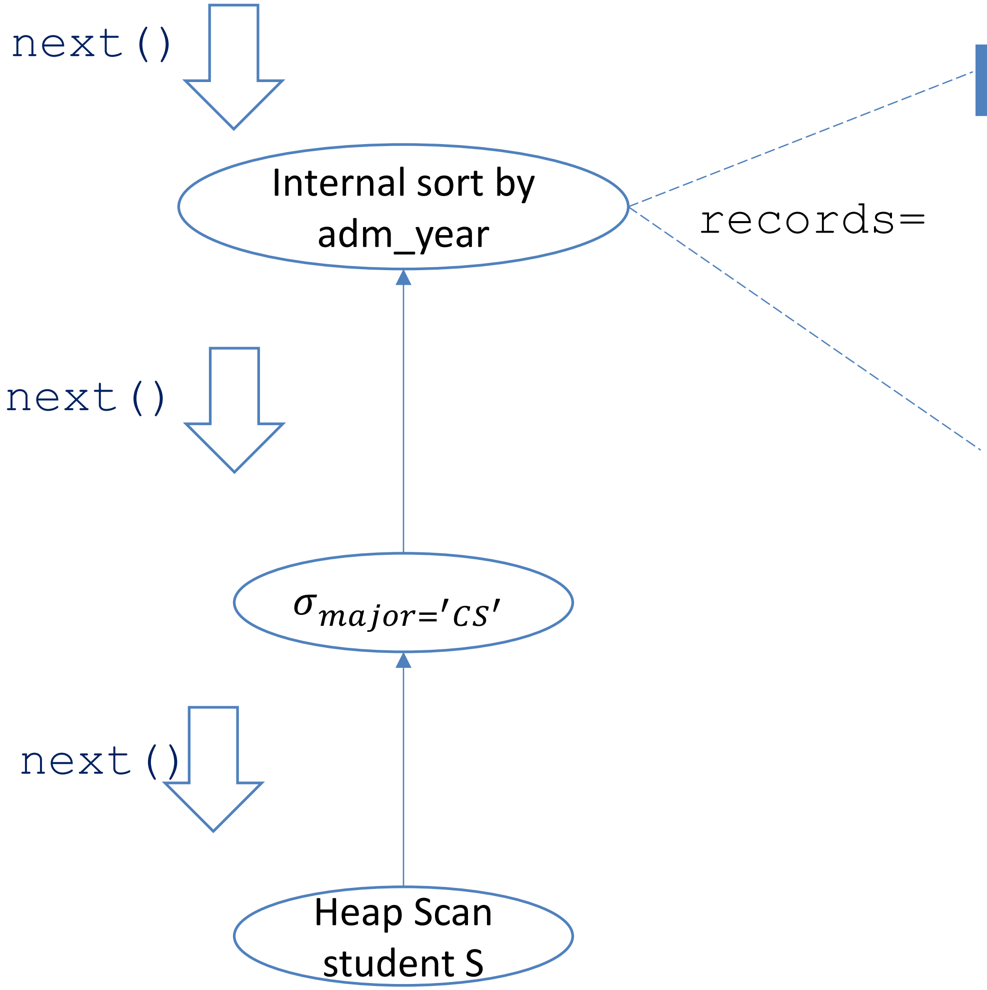
```
struct internal_sort_iterator: public iterator {// ctor omitted
    void init() {
        input[0]->init();   // iterator implementation must recursively initialize the inputs
    }
    Record next() {
        if (!valid) {
            while (r = input[0]->next()) records.push_back(r);
            sort r; set i to 0; set valid to true;
        } // will not return until all the records from the input are fetched
        if (i < records.size()) return records[i++];
        return an invalid record;
    }
    void close() {
        input[0]->close();
    }
    void rewind() {
        set i to 0;    // think: why not call input[0]->rewind()?
    }
   // internal state of an internal sort. note that all the records from the input iterator are stored here.
    Expressions *columns;
    int n;
    bool valid;
    size_t i;
    vector<Record> records;
};
```

Example: putting it together



sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

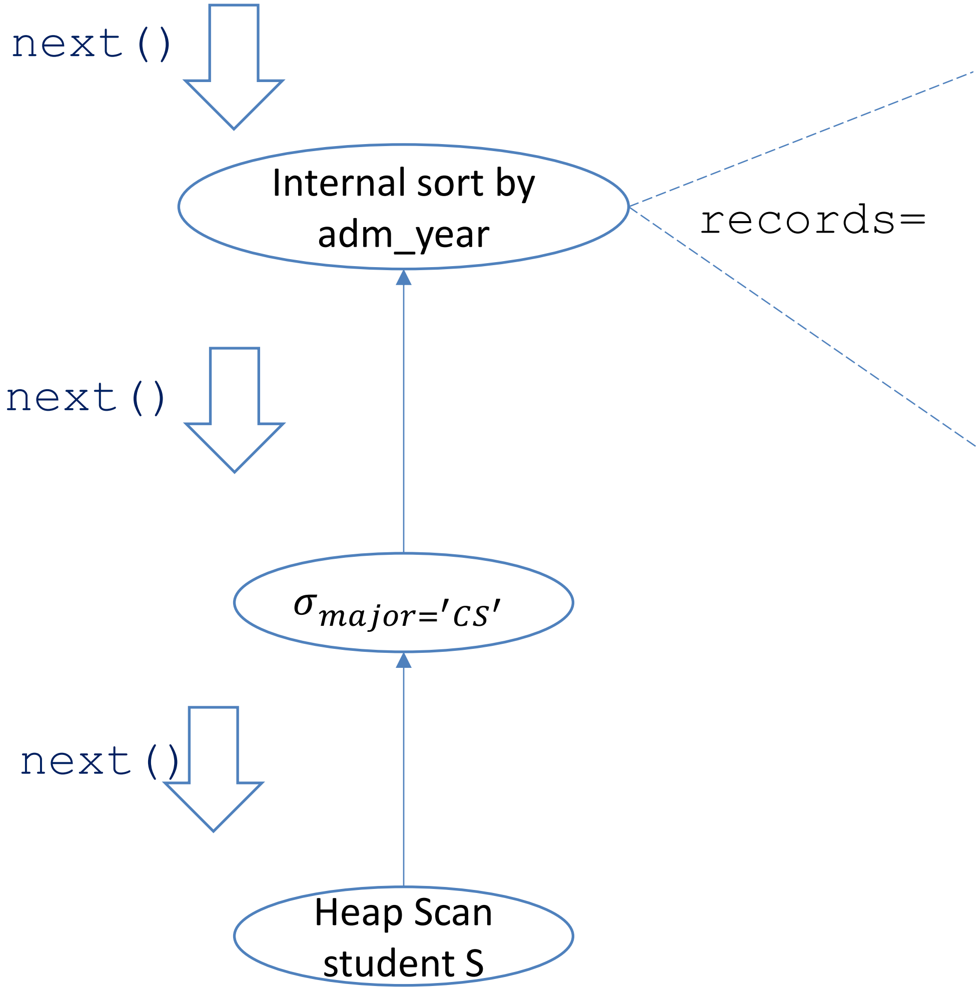
Example: putting it together



sid	name	login	major	adm_year

sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

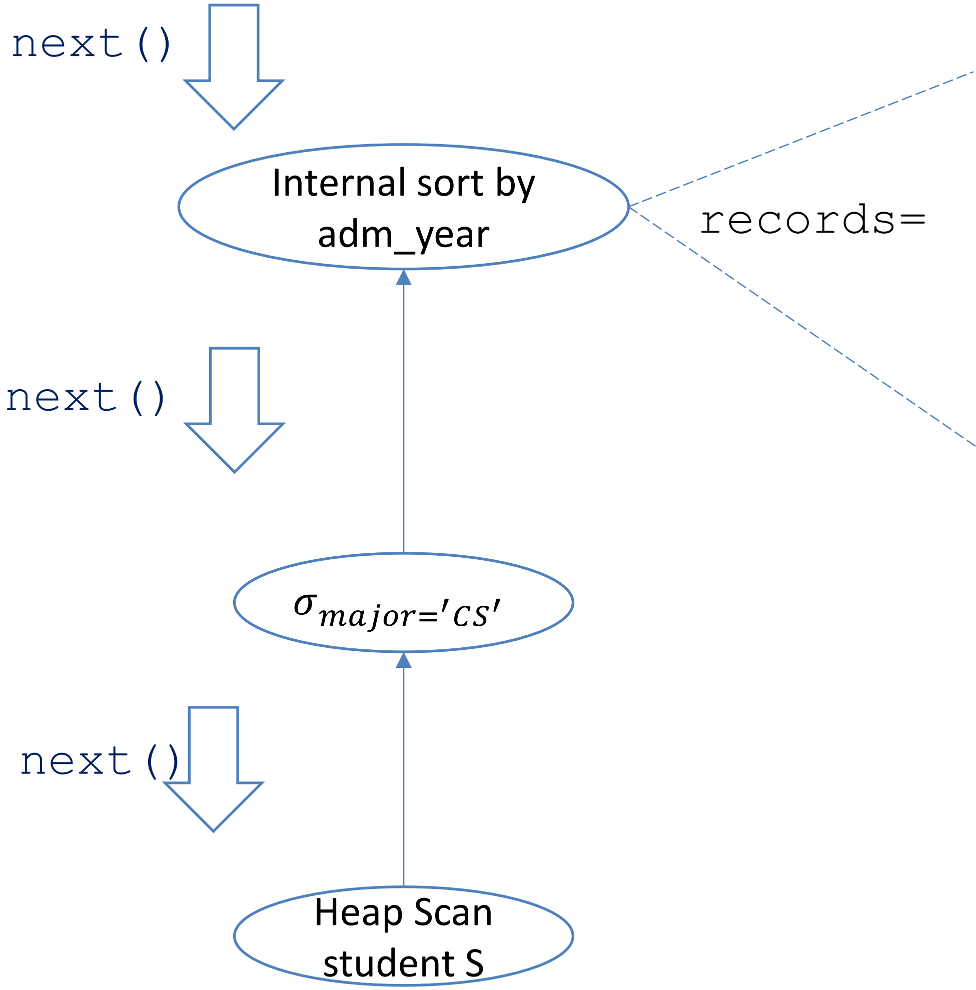
Example: putting it together



sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021

sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

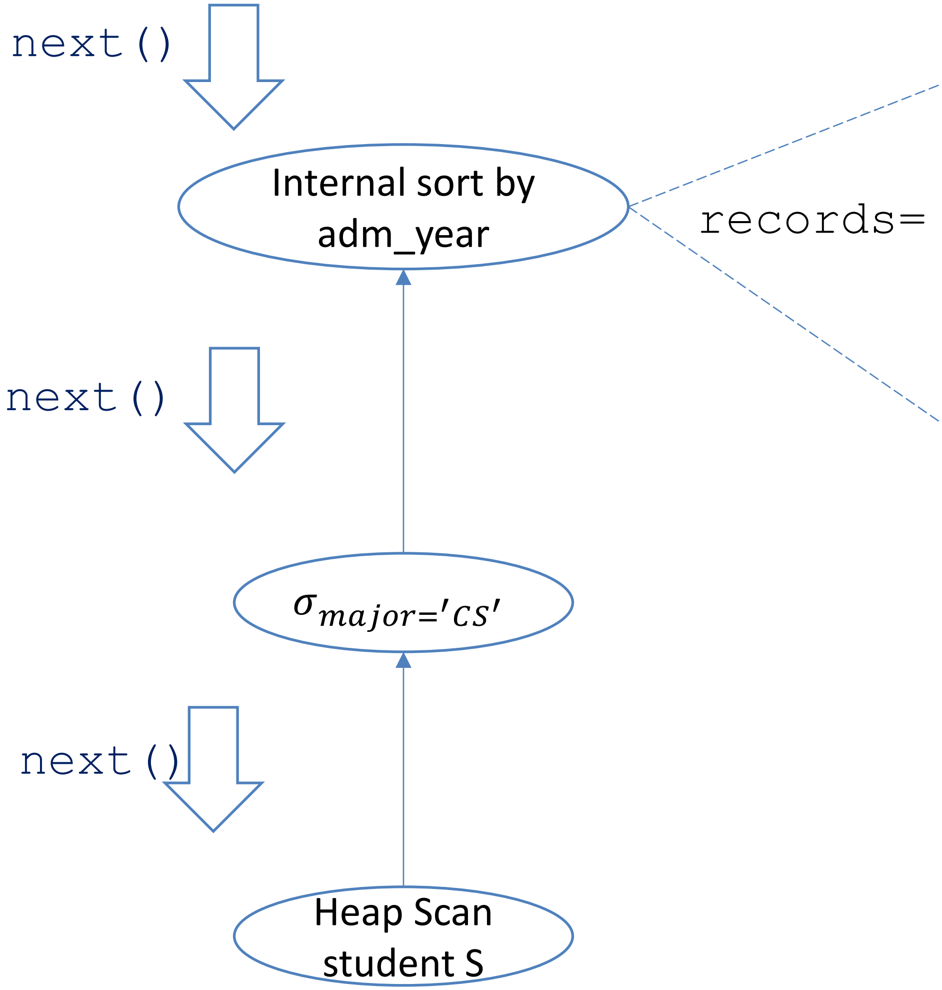
Example: putting it together



sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021

sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

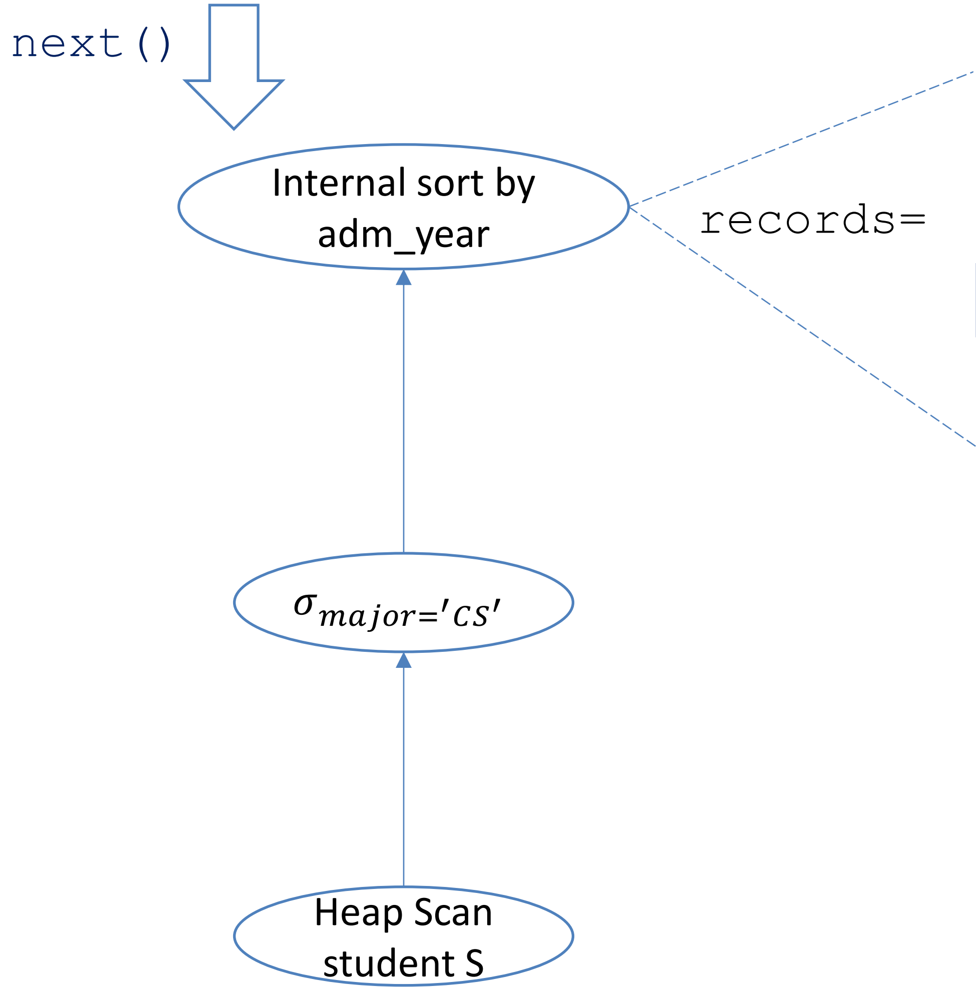
Example: putting it together



sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

Example: putting it together



sid	name	login	major	adm_year
103	David	davel	CS	2020
102	Charlie	charlie7	CS	2021
100	Alice	alicer34	CS	2021

sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

Materialization model

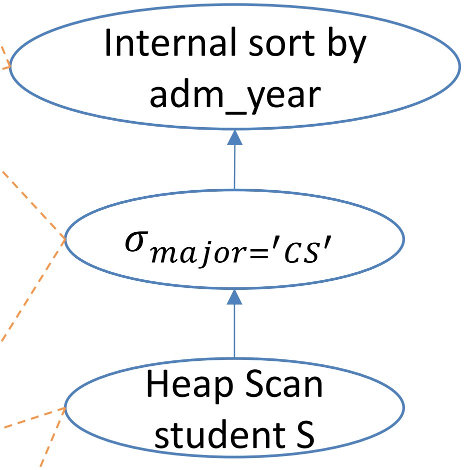
- Fully materializes results in each operator
  - Emits all results as a whole
  - Can send tuples in row or column formats
  - Can push down hints to avoid scanning too many records

- Good for queries that touches a few records at a time
  - OLTP workload
  - Not good for those with large intermediate results

```
output = child.output()
sort(output)
return out
```

```
out = []
for t in child.output():
  if t.major = ‘CS’:
     out.append(t)
return out
```

```
out = []
for t in S:
  out.append(t)
return out;
```



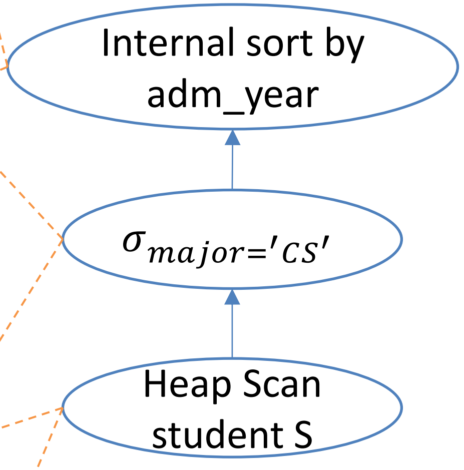
Vectorization model

- Emits a small batch of results at a time
  - Still needs to loop over a `next()` function
  - Fewer function calls & can often leverage SIMD
  - Bounded memory usage unlike materialization model
  - Good for OLAP workload
  - Batch size may depend on hardware or workload properties
- DBMS often takes a hybrid approach

```
out = []
while c_out = child.Next():
   out.extend(c_out)
sort(out)
return out
```

```
out = []
while c_out = child.Next():
   out.extend(
      filter(c_out, “major = ‘CS’”))
   if |out| >= k:
      return out
```

```
out = []
continue scan t in S:
   out.append(t)
   if |out| >= k:
      return out
```


Internal sort by adm_year
$\sigma_{major='CS'}$
Heap Scan student S