# CSE462/562: Database Systems (Spring 24)

## Lecture 8: Query Processing Overview

## 3/25/2024

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

Last updated: 3/19/2024

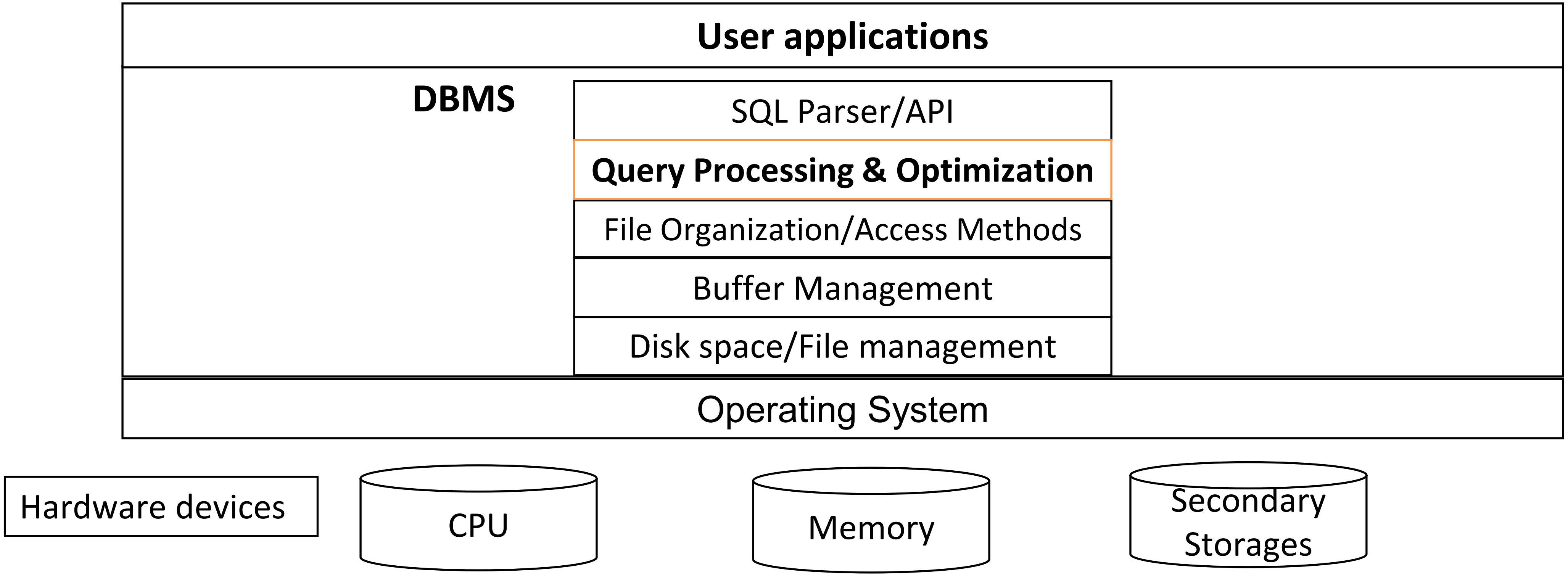
# Midterm review & Q&A

- Reminders
  - Midterm exam on 3/27/2024, Knox 104, 7:05 pm - 8:25 pm
    - Open-book, paper materials only, no electronics except a calculator
    - Please arrive at least 5 minutes early
    - Bring your ID

- Covers everything up to Lecture 6
  - Excluding relational model, relational algebra & SQL

- The lecture on 4/8 will be remote due to the solar eclipse
  - Live streaming from Knox 104
  - Please join through Panopto
    - https://ub.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=d5b61364-381a-4596-8f26-b0f20148c17a

# Big picture


User applications
DBMS
SQL Parser/API
Query Processing & Optimization
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary Storages

# What's discussed so far

- The lower-level storage layer in DBMS
  - Disk/file space management
  - Buffer management
  - File organization
  - Access methods
    - Indexing
- How to answer queries/perform updates?
  - Relational algebra vs SQL
  - *Correctness?*
  - *Efficiently?*
- Query processing & optimization

student

| 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 |

Find the names and the grades of all the students enrolled in the course 562 who were admitted in the year of 2021?

| | | | |
|---|---|---|---|
| 100 | s22 | 562 | 4.0 |
| 102 | s22 | 562 | 2.3 |
| 100 | f21 | 560 | 3.7 |
| 101 | s21 | 560 | 3.3 |
| 102 | f21 | 560 | 4.0 |
| 103 | s22 | 460 | 2.7 |
| 101 | f21 | 560 | 3.3 |
| 103 | f21 | 250 | 4.0 |

# Simple select query and relational algebra

- Recall that the basic form of SELECT query can be translated into extended relational algebra
  - The conceptual way of answering the query
  - With some non-relational operators (notably Sort).

**-- SQL SELECT with no aggregation**

SELECT [DISTINCT] $E_1, E_2, \ldots, E_m$
FROM $R_1, R_2, \ldots, R_n$
[WHERE $P$]
[ORDER BY *expr* [ASC|DESC] [,…]]

non-relational

$$Sort\left(Distinct\left(\pi_{E_1,E_2,\ldots E_m}\sigma_P R_1 \times R_2 \times \cdots \times R_n\right)\right)$$

**-- SQL with aggregation**

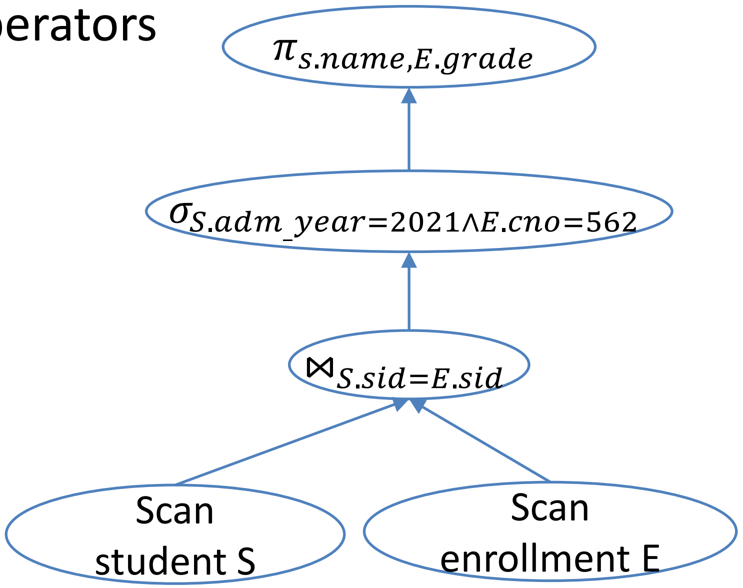
SELECT $E'_1, E'_2, \ldots, E'_m, F_1(E''_1), \ldots, F_k(E''_k)$
FROM $R_1, R_2, \ldots, R_n$
[WHERE $P$]
[GROUP BY $E_1, E_2, \ldots, E_l$
[HAVING $P'$]]
[ORDER BY expr [ASC|DESC] [,…]]

$$Q \leftarrow \sigma_P R_1 \times R_2 \times \cdots \times R_n$$

$$Sort\left(\pi_{E'_1,E'_2,\ldots,E'_m,F_1(E''_1),\ldots,F_k(E''_k)}\sigma_{P'}\left({}_{E_1,E_2,\ldots,E_l}\gamma_{F_1(E''_1),\ldots F_k(E''_k)}Q\right)\right)$$

# Query processing overview

- DBMS translates SQL to a special internal language
  - Query plans
    - *logical:* extended relational algebra with some non-relational operators
    - *physical:* describes the actual implementation of the operators
- Think of query plans as data-flow graphs
  - Edges: flow of records
  - Vertices: relational and non-relational *operators*
    - Input/Output of the operators: relations




An example of logical plan

- Three stages of query processing
  - **Parsing & query rewriting:** SQL -> logical plan
  - **Query optimization:**
    logical plan -> optimized logical plan -> physical plan
  - **Query execution:** evaluating the physical plan over the database

# Query processing overview

*ODBC/JDBC/ command line frontend*

**

**SQL Query**

```
SELECT S.name,E.grade
FROM student S, enrollment E
WHERE S.sid = E.sid
  AND S.adm_year = 2021
  AND E.cno = 562;
```

*SQL Parser**

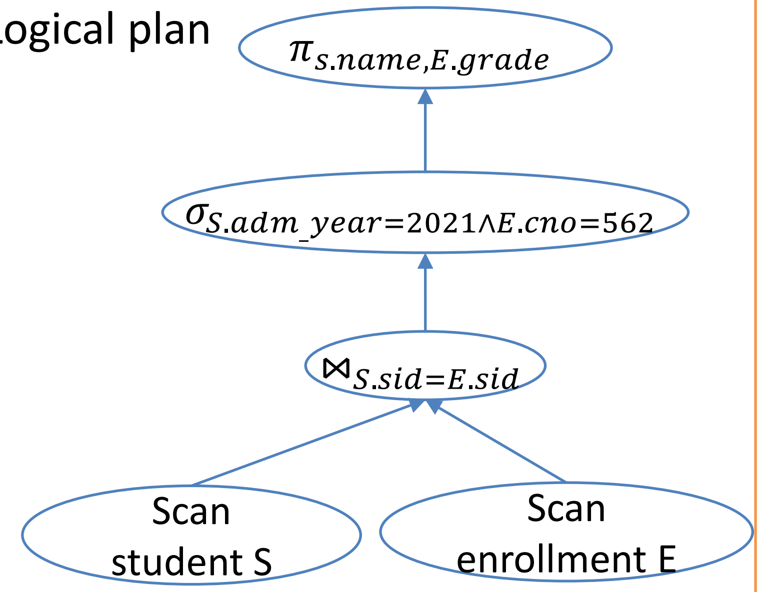
** include multiple intermediate steps (e.g., parsing tree/analysis/rewriting)*

**(Extended) Relational Algebra**

$\pi_{S.name,E.grade}\sigma_{S.adm\_year=2021 \wedge E.cno=562} S \bowtie_{S.sid=E.sid} E$

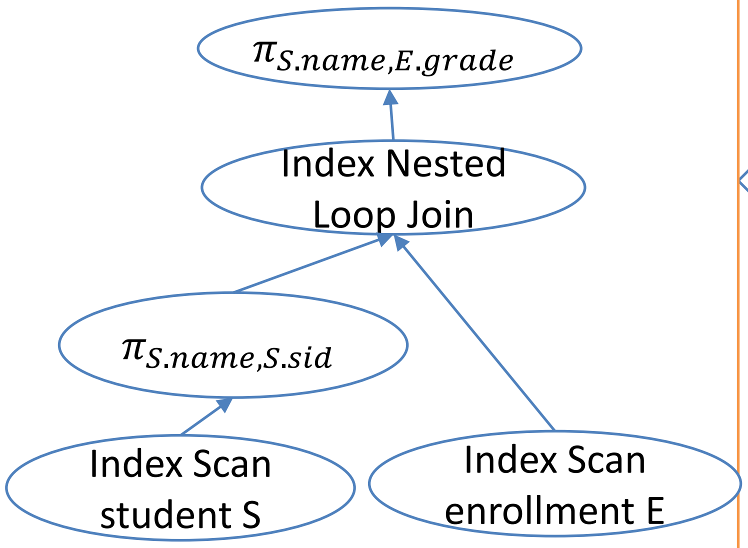
*Internally represented as*

**Logical plan**




*Query Optimizer*

**Physical plan**




*Query Execution*

**Query result**

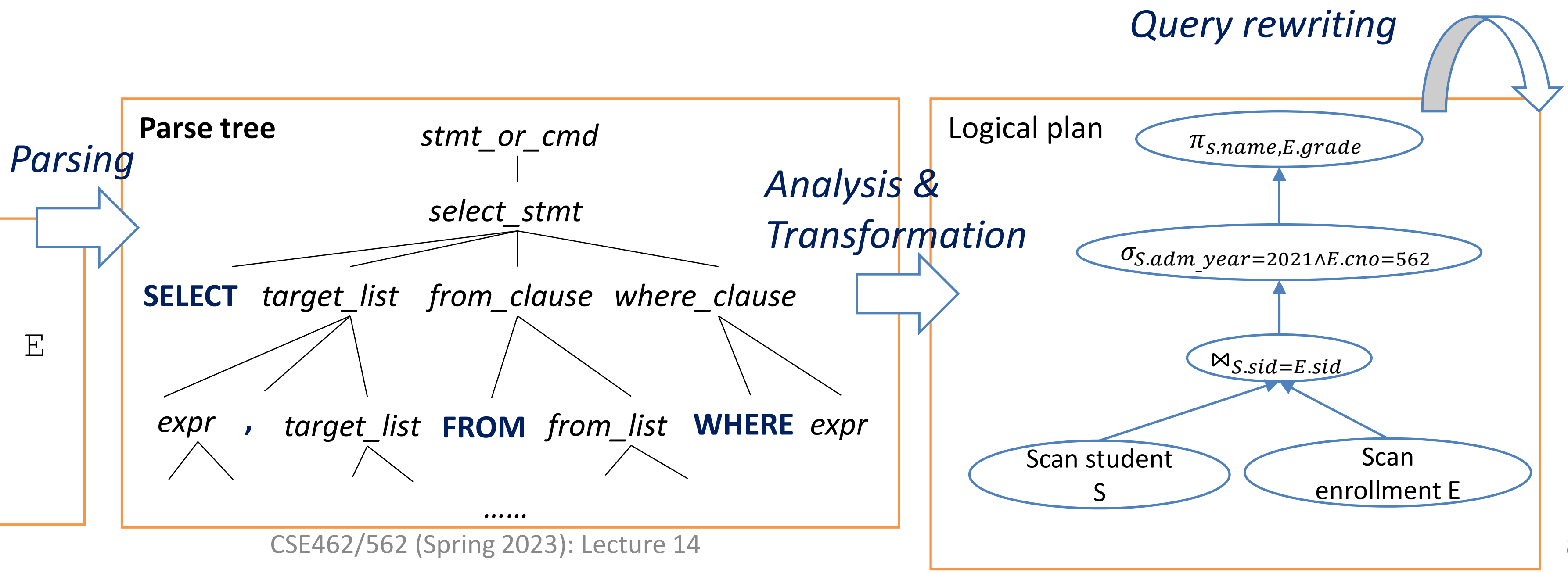
```
S.name  | E.grade
Alice   | 4.0
Charlie| 2.3
(2 rows)
```

# Parsing and query rewriting

- SQL Parser are usually generated from a context-free grammar using compiler tools
  - e.g., antlr (LL grammar), lex+yacc/flex+bison (LR grammar/LALR(1) grammar)
  - We'll omit the details which are covered in compiler courses
  - Produces a *parse tree* for a SQL query
- Analysis and transformation into logical plan
  - A parse tree represents the syntactical structure of a SQL query -- not suitable for query processing
  - Needs to be translated into a logical plan
  - Catalog information helps resolving tables/columns/types/expressions/functions
- Query rewriting
  - User defined/system defined rules for transforming queries (e.g., non-materialized views, customized rewriting rules)

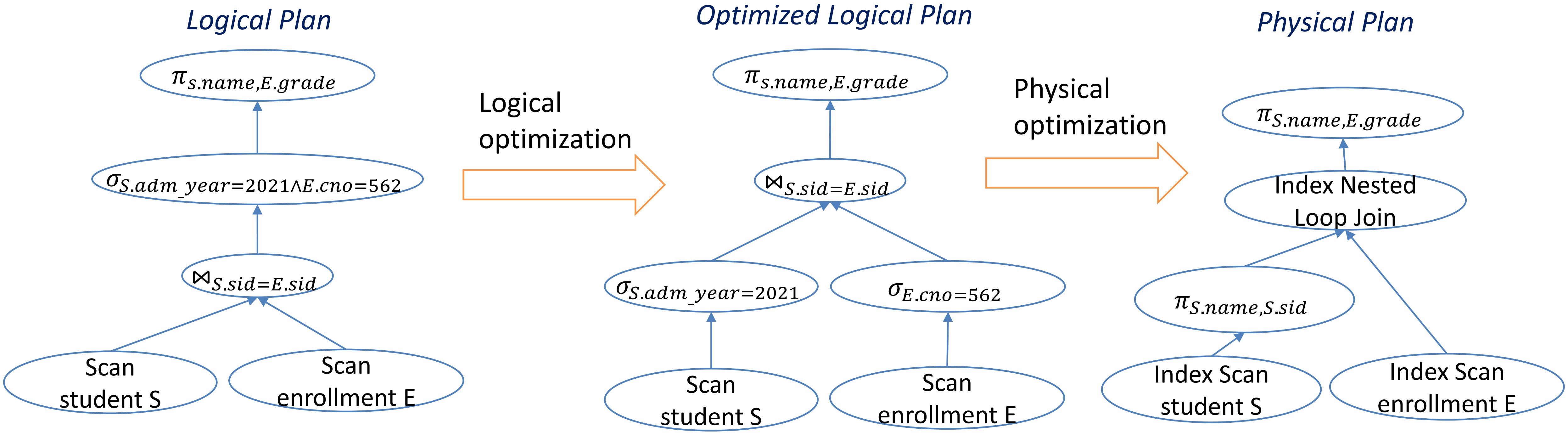
**SQL Query**

```
SELECT S.name, E.grade
FROM student S, enrollment E
WHERE S.sid = E.sid
  AND S.adm_year = 2021
  AND E.cno = 562;
```



# Query optimization (a preview)

- Many equivalent plans exist for the same query
  - Efficiency varies
- Query optimization
  - Finding ~~the best~~ *a not-too-bad plan* with reasonable overhead
  - Generally divided into two phases



# Query execution

- Query executor needs to evaluate the result of a physical plan over a database instance

- Query interpretation vs compilation
  - To date, most DBMS uses a single piece of binary code that "interprets" the query plans
    - Uses run-time information to determine which function(s) to call
    - Easy to implement with runtime polymorphism (e.g., C++/Java/Scala)
  - Some modern DBMS compiles query plans into binary code for efficiency (e.g., [1])
    - Avoids virtual function call overhead in tight loops
    - More efficient for queries over large database
    - Overhead for compilation (LLVM to the rescue) and a bit harder to implement

  - Can take hybrid approach:
    - e.g., only compiling expression trees into binary code, while interpreting the physical plan

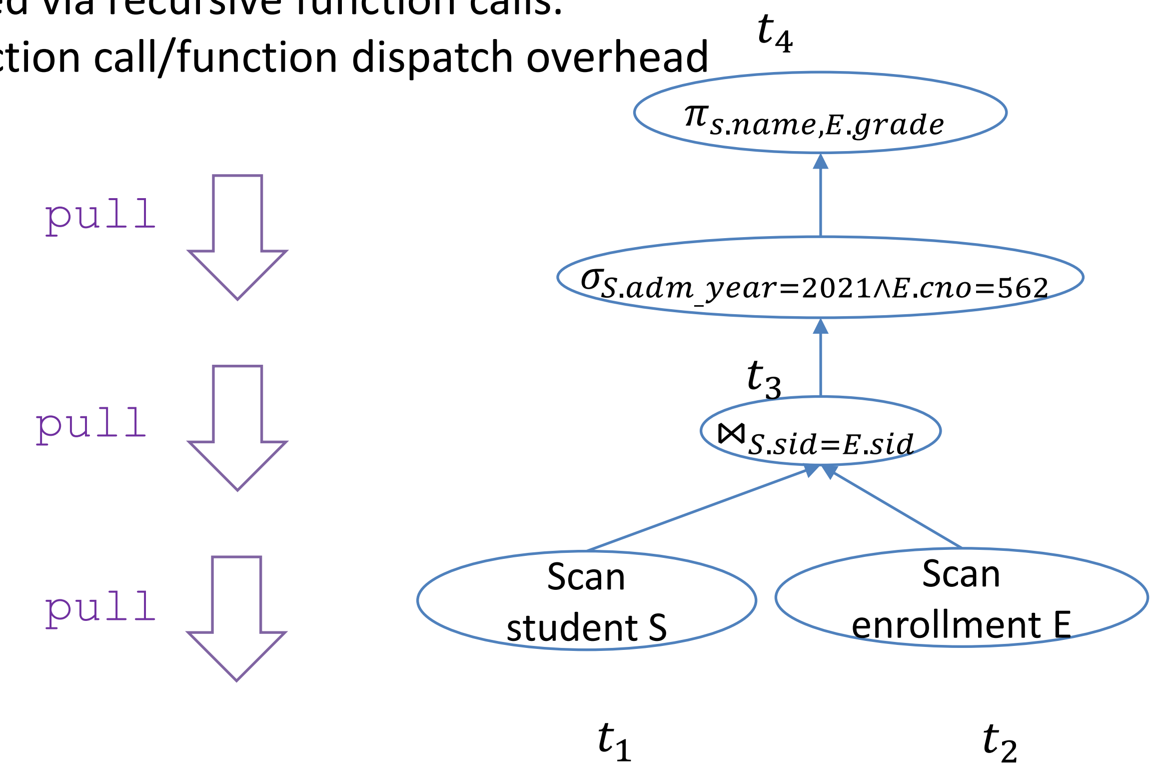
[1] Efficiently Compiling Efficient Query Plans for Modern Hardware. Thomas Neumann, 2011.

# Query execution (cont'd)

- *Pull-based* vs *push-based* query execution

**Pull-based query execution**

- Start from root and pull data from children
- Tuple passed via recursive function calls.
- Virtual function call/function dispatch overhead

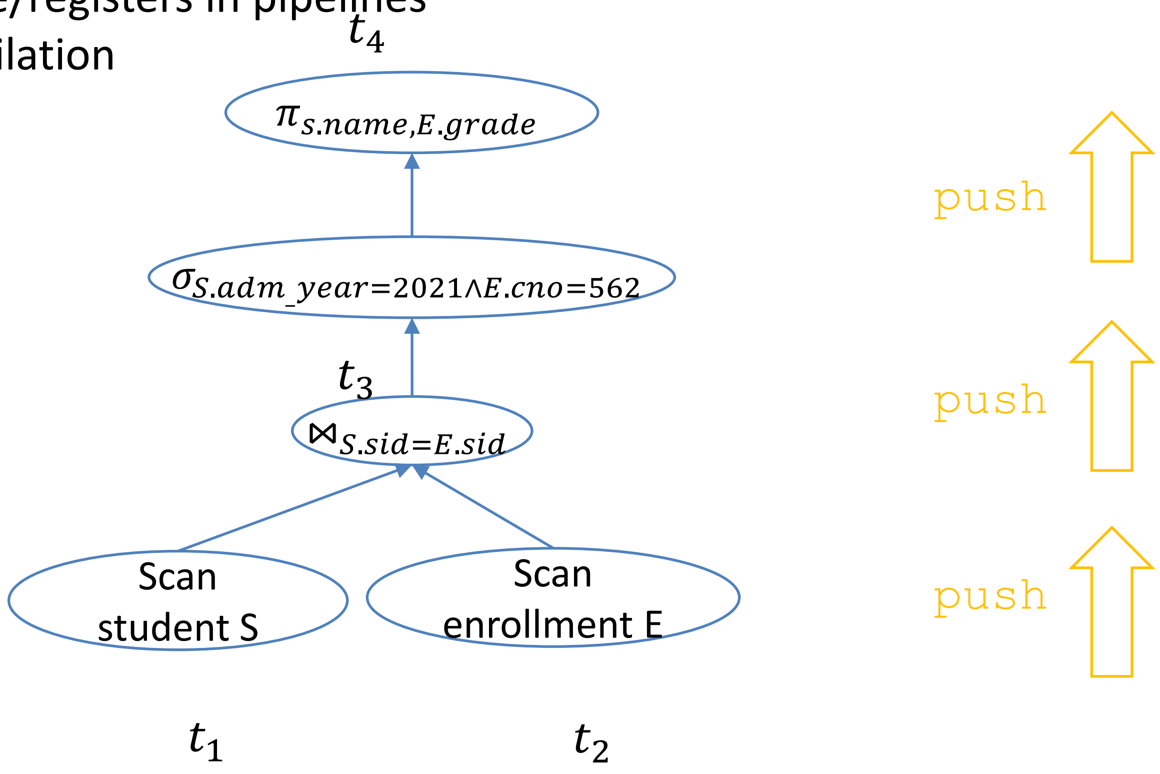


# Query execution (cont'd)

- *Pull-based* vs *push-based* query execution

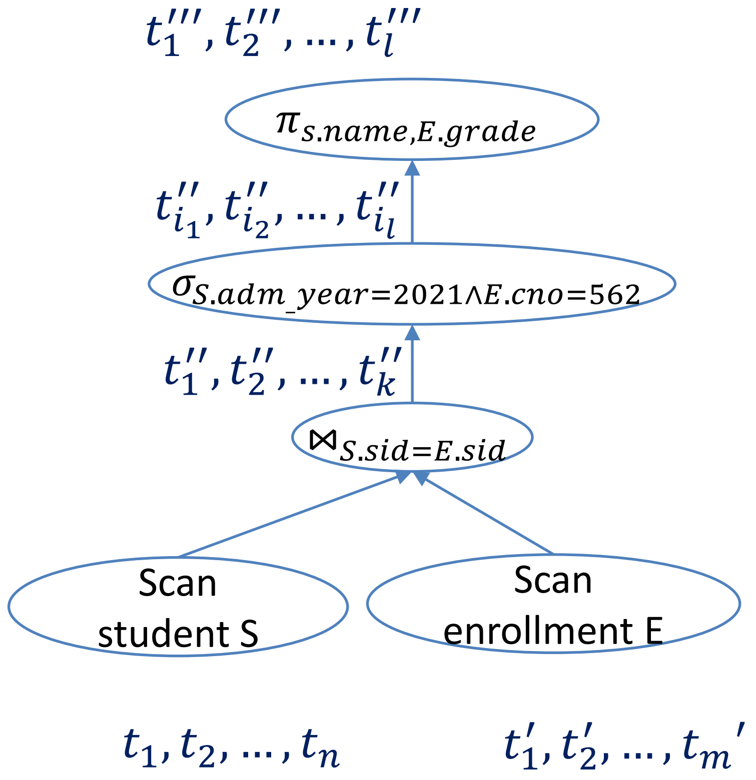
**Push-based query execution**

- Start from leaf and push data to parent
- Allows more efficient use of cache/registers in pipelines
  - when used with query compilation



# Query execution (cont'd)

- *Pull-based* vs *push-based* query execution
- *Pipelining* vs *materialization*

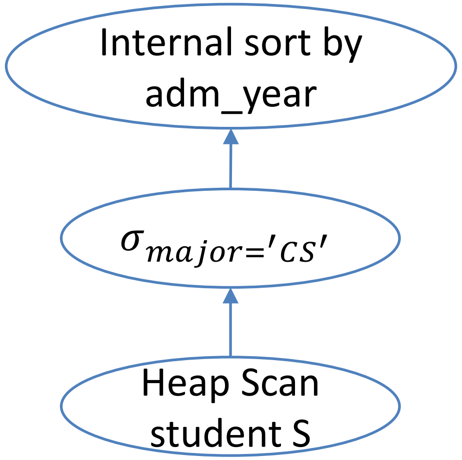


# Query execution models

- 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;
};
```

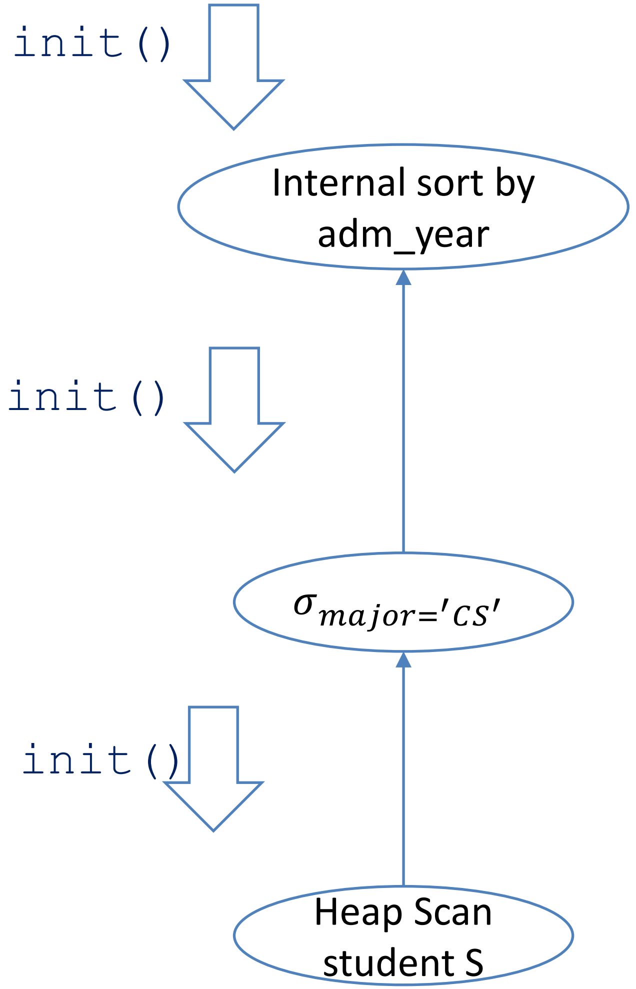
# Example: selection σ (streaming)

```
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)

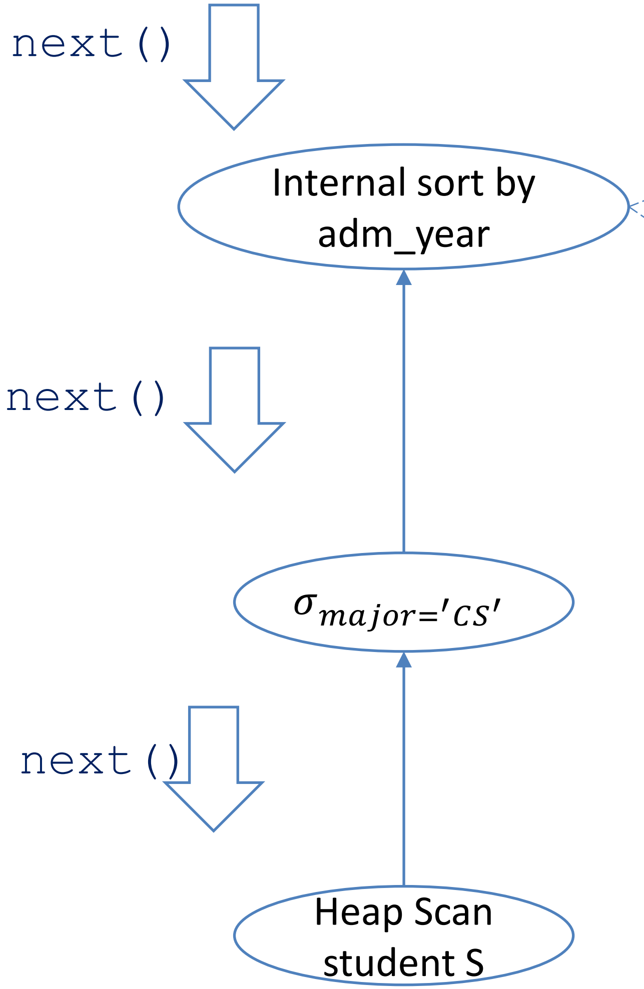
```
struct internal_sort_iterator: public iterator {// ctor omitted
    void init() {
        input[0]->init();    // iterator implementation must recursively initialize the inputs
    }
    Record next() {
        if (!valid) {
            while (r = input[0]->next()) records.push_back(r);
            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

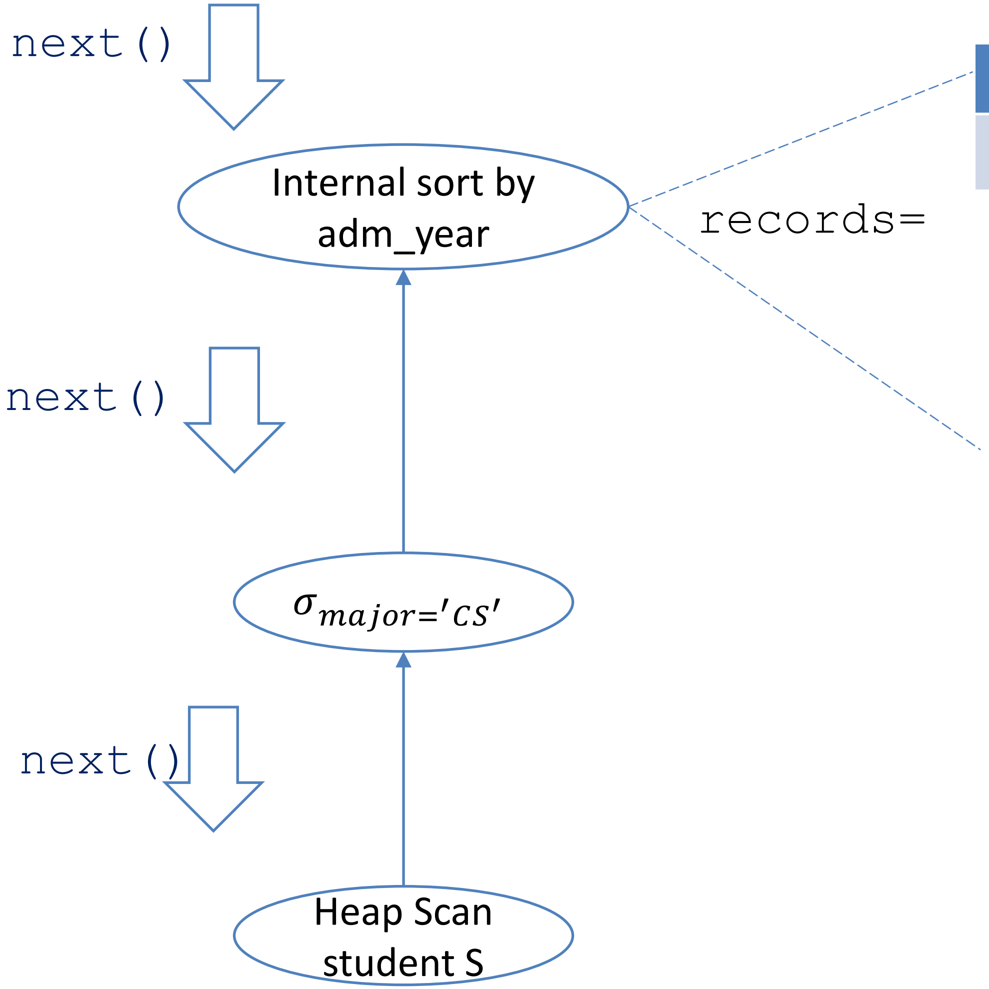



records=

| 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 |

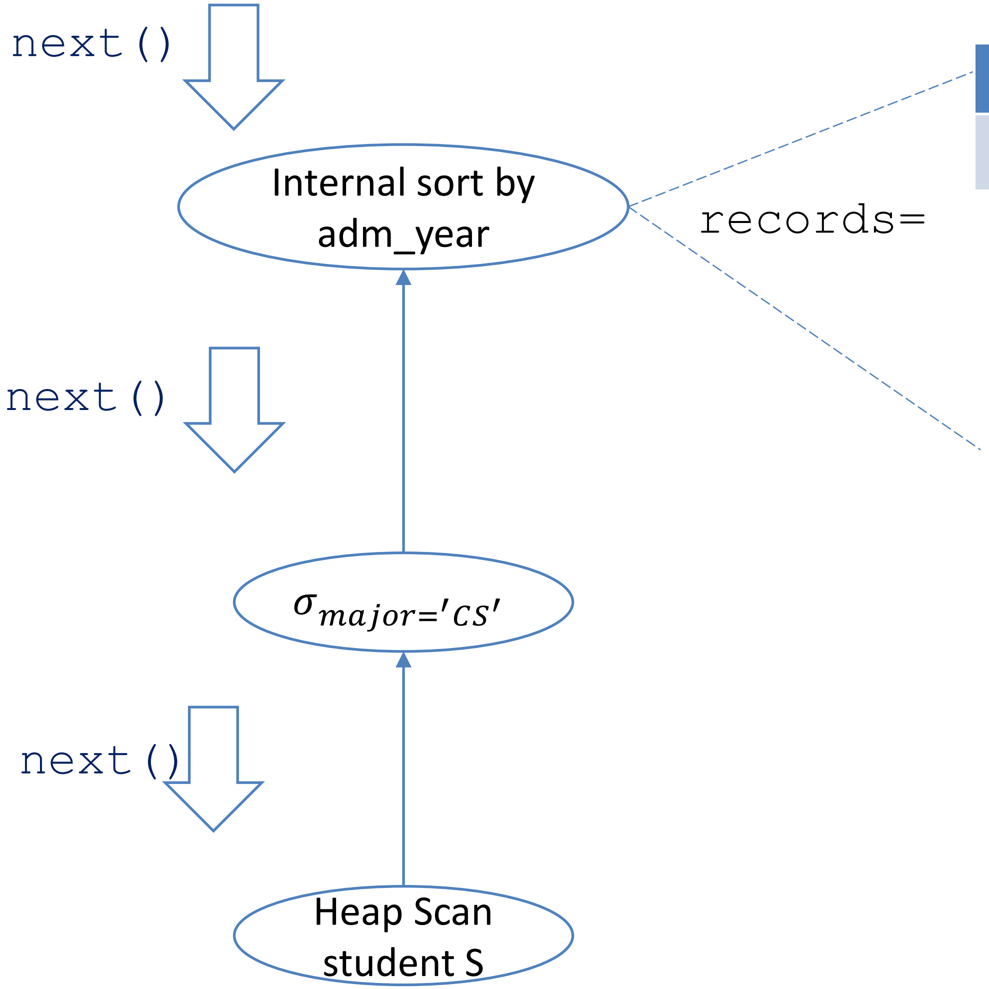
# Example: putting it together




| 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 |

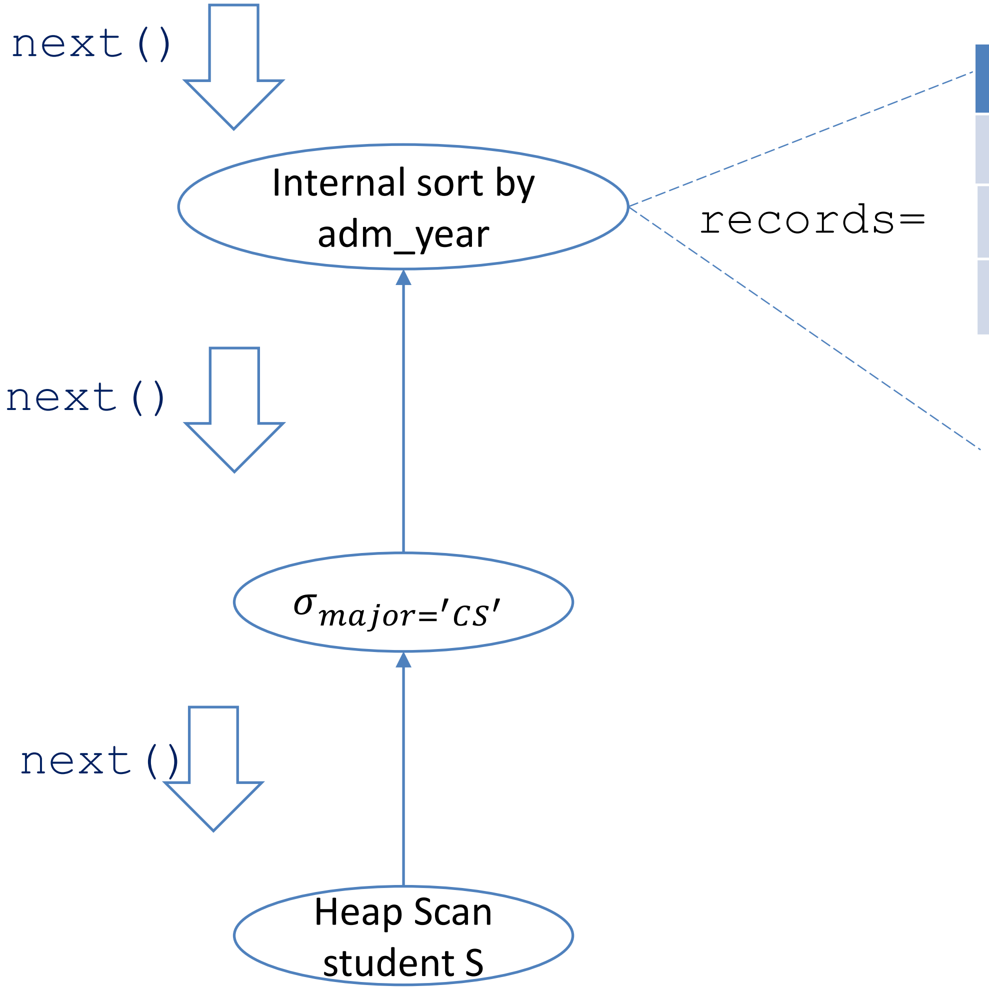
# Example: putting it together




| 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 |

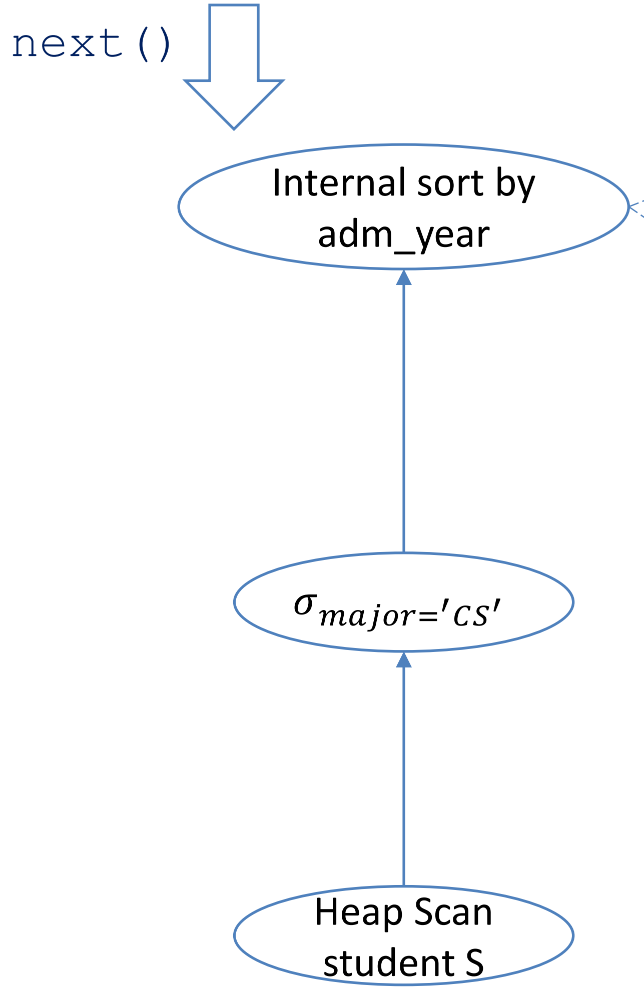
# Example: putting it together




| 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




records=

| 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

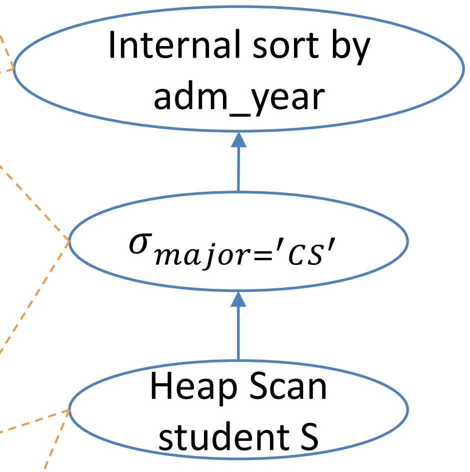
- Fully materializes results in each operator
  - Emits all results as a whole
  - Can send tuples in row or column formats
  - Can push down hints to avoid scanning too many records

- 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;
```



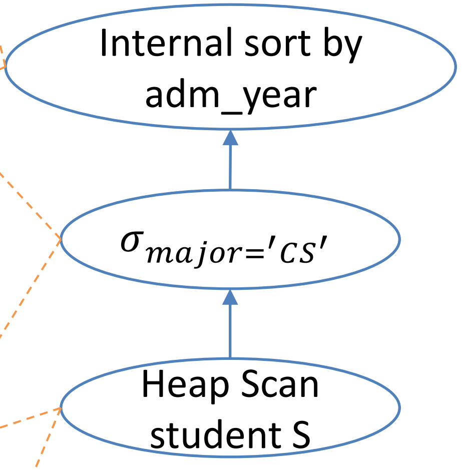
# Vectorization model

- 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
```



# Summary

- This lecture
  - Overview of query processing
  - Query execution models

- Next two lectures
  - Single-table query processing

- Reminders
  - Midterm exam on 3/27/2024, Knox 104, 7:05 pm - 8:25 pm
    - Open-book, paper materials only, no electronics except a calculator
    - Please arrive at least 5 minutes early
    - Bring your ID
  - The lecture on 4/8 will be remote due to the solar eclipse
    - Live streaming from Knox 104
    - Please join through Panopto
    - https://ub.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=d5b61364-381a-4596-8f26-b0f20148c17a