CSE 707SEM: Select Topics on Modern Database Systems (Fall 22)

Lecture 0: Introduction

8/31/2022

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

About me

- Instructor: Zhuoyue Zhao, zzhao35@buffalo.edu
- Office: Davis Hall 338I
- Course website: https://cse.buffalo.edu/~zzhao35/teaching/cse707_fall22/
- No fixed office hour
  - Send me a message on Piazza to schedule one on demand

Course description

- Davis 113A
- Every Wednesday 10:00 AM - 12:50 PM
  - We are likely to end early if there's only one presentation for that week
- No textbook required
  - But you're required to read the papers
- Discussion and communication: https://piazza.com/buffalo/fall2022/cse707/home
- Assignment submission: https://ublearns.buffalo.edu/ultra

- Topics for Fall 2022:
  - OnLine Analytical Processing (OLAP) / Exact Queries
  - Approximate Query Processing

Requirements

- We will have up two paper presentation by students each time
- For student who is presenting
  - Start early on the presentation (at least one week in advance)
  - Send me the presentation slides by the Tuesday 10 AM prior to your presentation (you're encouraged to submit the first draft as early as possible)
    - I will send you comments on the slides
    - Please make yourself available for revising the presentation slides and resubmit before presentation
  - The presenter does not need to submit questions and summary for the paper presented
- For all students
  - Read the paper before lecture
  - Submit three questions for **each paper** you read by Tuesday 10 AM
  - Participate in discussion
    - random quizzes throughout the semester
  - Submit a short paper summary for **each paper** by Friday 10 AM
    - For weeks with two presentations, the deadline will be extended to Saturday 11:00 PM

Grading

- Grading items
  - 8% for the pre-lecture questions (0.5% each)
  - 32% for the paper summaries (2% each)
  - 30% for the presentation
  - 30% for participation
    - Based on your attendance, in-class discussion and random quizzes
    - Graded at the end of semester

- Satisfactory: >= 75%; Unsatisfactory: < 75%

Course schedule

Course Schedule

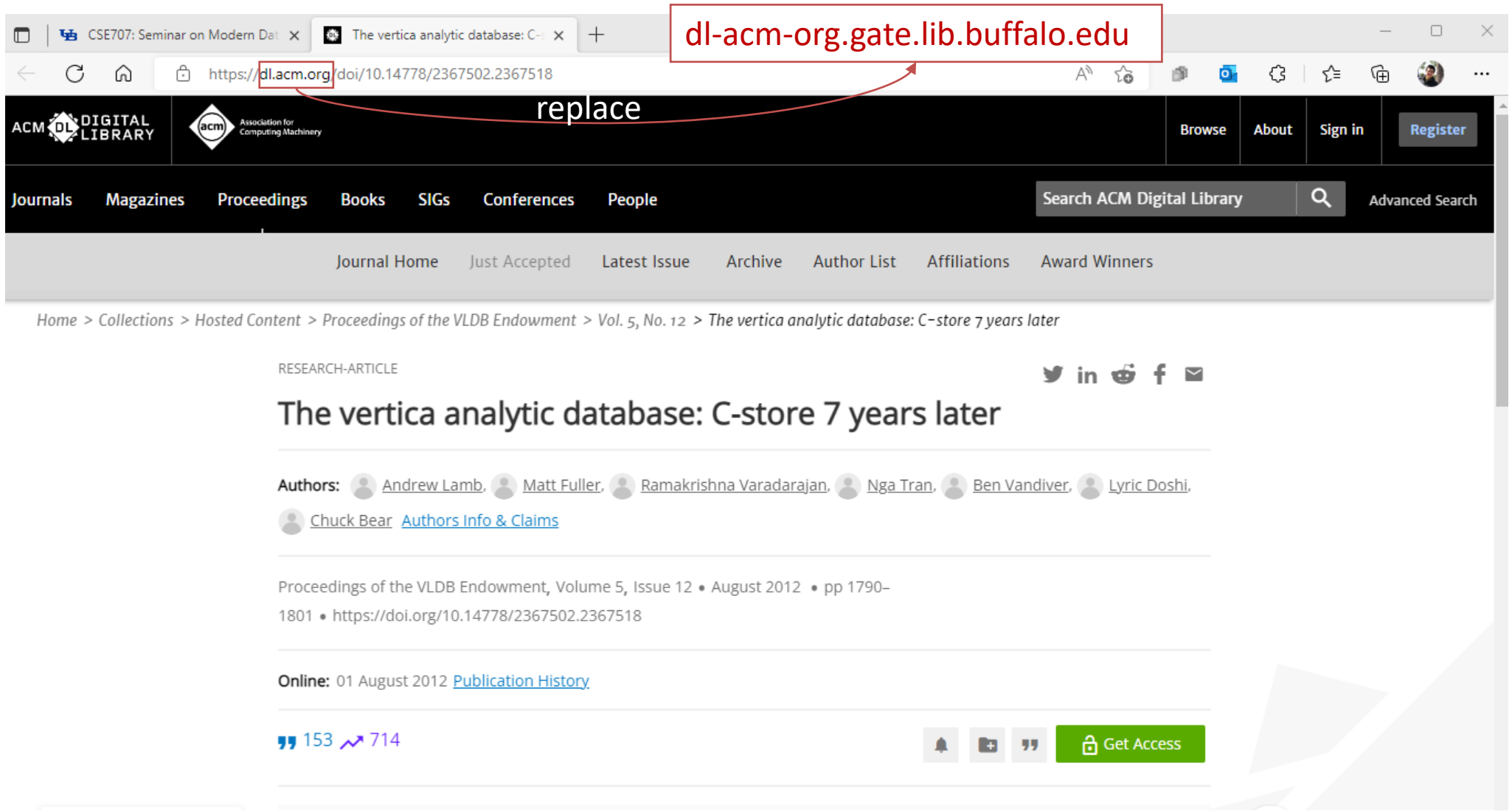
For UB students: Some of the following links may require an ACM Digital Library subscription. UB library has a paid subscription available for all students so you **do not** need to pay. You may either connect to eduroam when you're on campus, or replace `dl.acm.org` with `dl-acm-org.gate.lib.buffalo.edu` and enter your UBIT login credentials when you're off campus.

* Paper summary deadline is extended to Saturday at 11:00 PM for the weeks with two presentations.

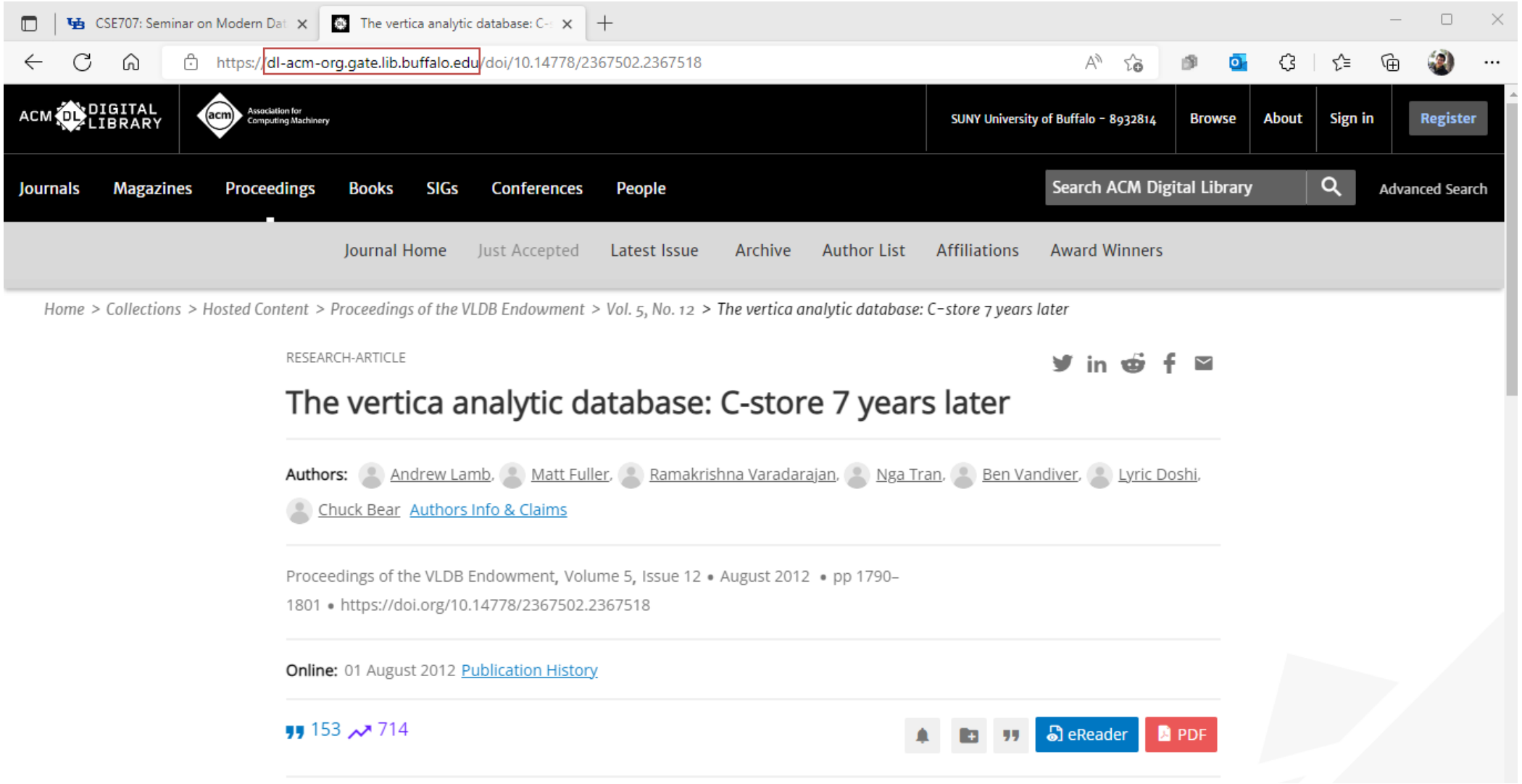
The following schedule is subject to change due to student add/drop. Please double check the latest schedules when completing your assignments.

Date	Topic	#	Required Readings	Presenter
8/31/2022	Logistics and Introduction			N/A
			Online Analytical Query Processing	
9/7/2022	Columnar store	10	Andrew Lamb, et al. The Vertica Analytic Database: CStore 7 Years Later. In VLDB '12	
9/14/2022	Vectorized query execution	20	Peter Boncz, et al. MonetDB/X100: Hyper-Pipelining Query Execution. In CIDR '05.	
9/21/2022	Query compilation	30	Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In VLDB '11.	
9/28/2022	SIMD	40	Orestis Polychroniou, et al. Rethinking SIMD Vectorization for In-Memory Databases. In SIGMOD '15.	
			Approximate Query Processing	
***10/5/2022**	Online Aggregation	50	Joseph M. Hellerstein, et al. Online aggregation. In SIGMOD '97.	
	Confidence Intervals	60	Peter J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In SSDBM '97.	
***10/12/2022**	Random Sample	70	Frank Olken. Random Sampling From Databases. Section 2.4 - 2.6; and	

How to access ACM Digital Library off campus



How to access ACM Digital Library off campus



Academic Integrity Policies

Academic Integrity

All assignments (pre-lecture questions, assignments and presentation slides) must be prepared and written independently and reflect the student's own opinions. Simply paraphrasing other students' work is considered as plagiarism and we take the recommended actions for any discovered academic integrity violation per Departmental and University policies, including receiving an F grade or other appropriate penalties depending on the severity of the violation. While discussion among students prior to and after lectures are allowed, it is your responsibility to ensure that your submission is not substantially similar to any other student's submission. Note that it is generally acceptable to use part or all of the presentation slides found on conference website or the author's website, as long as there are proper citations and acknowledgments.

Accessibility Resources

- Accessibility Resources. If you have any disability which requires reasonable accommodations to enable you to participate in this course, please contact the Office of Accessibility Resources in 60 Capen Hall, 716-645-2608 and the instructor of this course as soon as possible. The office will provide you with information and review appropriate arrangements for reasonable accommodations, which can be found on the web at: http://www.buffalo.edu/studentlife/who-weare/departments/accessibility.html.

Today's agenda

- Quick recap on query processing basics

- Sign up for paper presentations

Relational model

Database schema

student(sid: integer, name: string, login: string, major: string, adm_year: date)

enrollment(sid: integer, semester: string, cno: integer, grade: float)

Relation (schema)

Relation (instance)

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

Record

Column

enrollment

sid	semester	cno	grade
100	s22	562	2.0
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

Database instance

Relational algebra

- There are 6 basic operators and commonly used compound operators:
  - Selection σ
  - Projection π
  - Renaming ρ
  - Cartesian product $\times$
  - Set difference $-$
  - Union $\cup$
  - Join $\bowtie$
    - Inner join
      - Natural join
    - Outer join
  - Set intersection $\cap$
  - Division operator /

- The operators takes relations as input, and outputs a relation
  - Schemas of the input/output schema are fixed
  - Operators can be composed
- https://cse.buffalo.edu/~zzhao35/teaching/cse562_spring22/files/03-rm.pdf

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

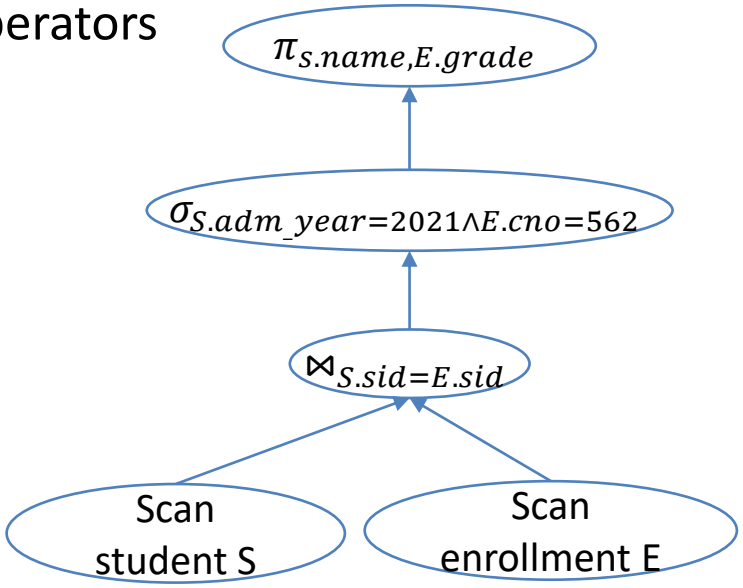
SELECT $E'_1, E'_2, \ldots, E'_m, F_1(E''_1), \ldots, F_k(E''_k)$
FROM $R_1, R_2, \ldots, R_n$
[WHERE P]
[GROUP BY $E_1, E_2, \ldots, E_l$
[HAVING P']]
[ORDER BY expr [ASC|DESC] [,…]]

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

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

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
- 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



An example of logical plan

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**

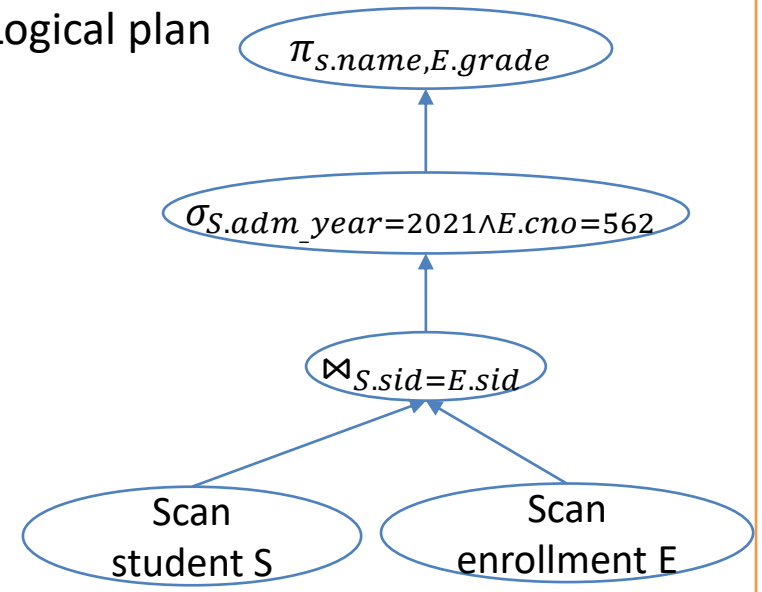
** include multiple intermediate steps (e.g., parsing tree/analysis/rewriting)*

(Extended) Relational Algebra

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

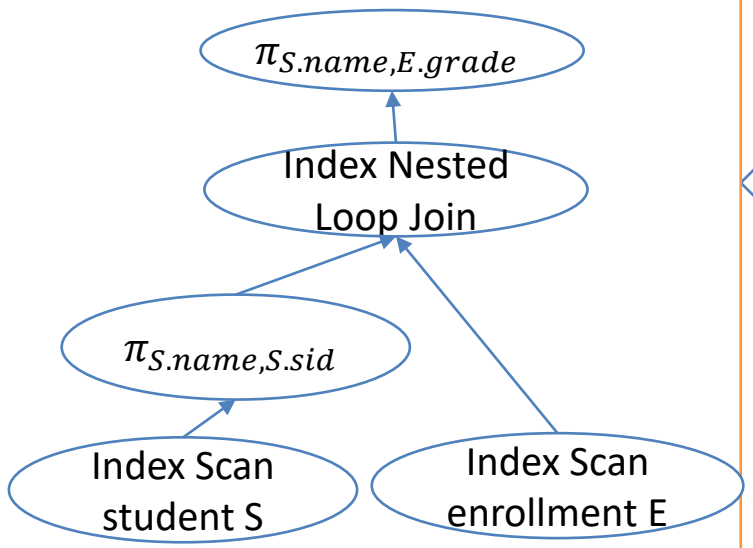
Internally represented as

Logical plan



Query Optimizer

Physical plan



Query Execution

Query result

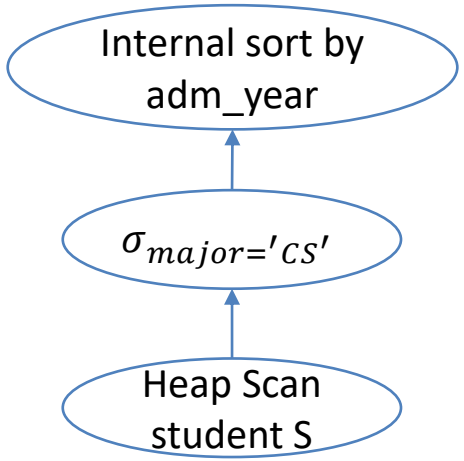
```
S.name  | E.grade
Alice   | 4.0
Charlie | 2.3
(2 rows)
```

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;
};
```

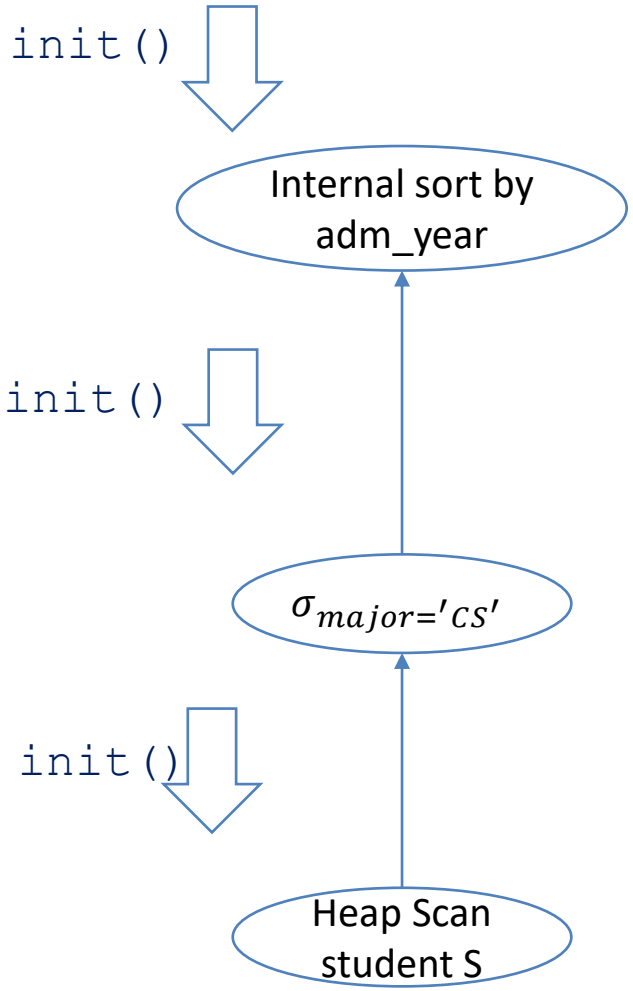
Example: selection σ (streaming)

```
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)

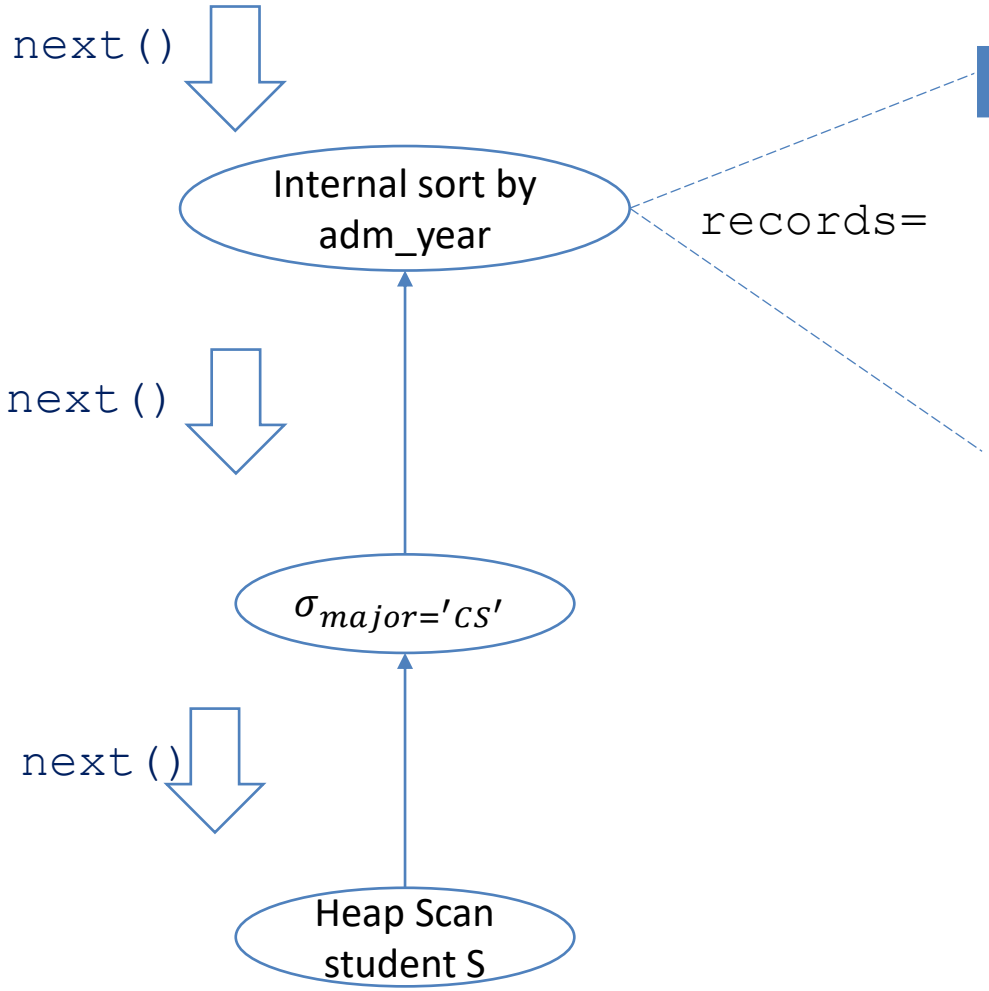
```
struct internal_sort_iterator: public iterator { // ctor omitted
    void init() {
       input[0]->init();   // iterator implementation must recursively initialize the inputs
    }
    Record next() {
       if (!valid) {
          while (r = input[0]->next()) records.push_back(r);
          sort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



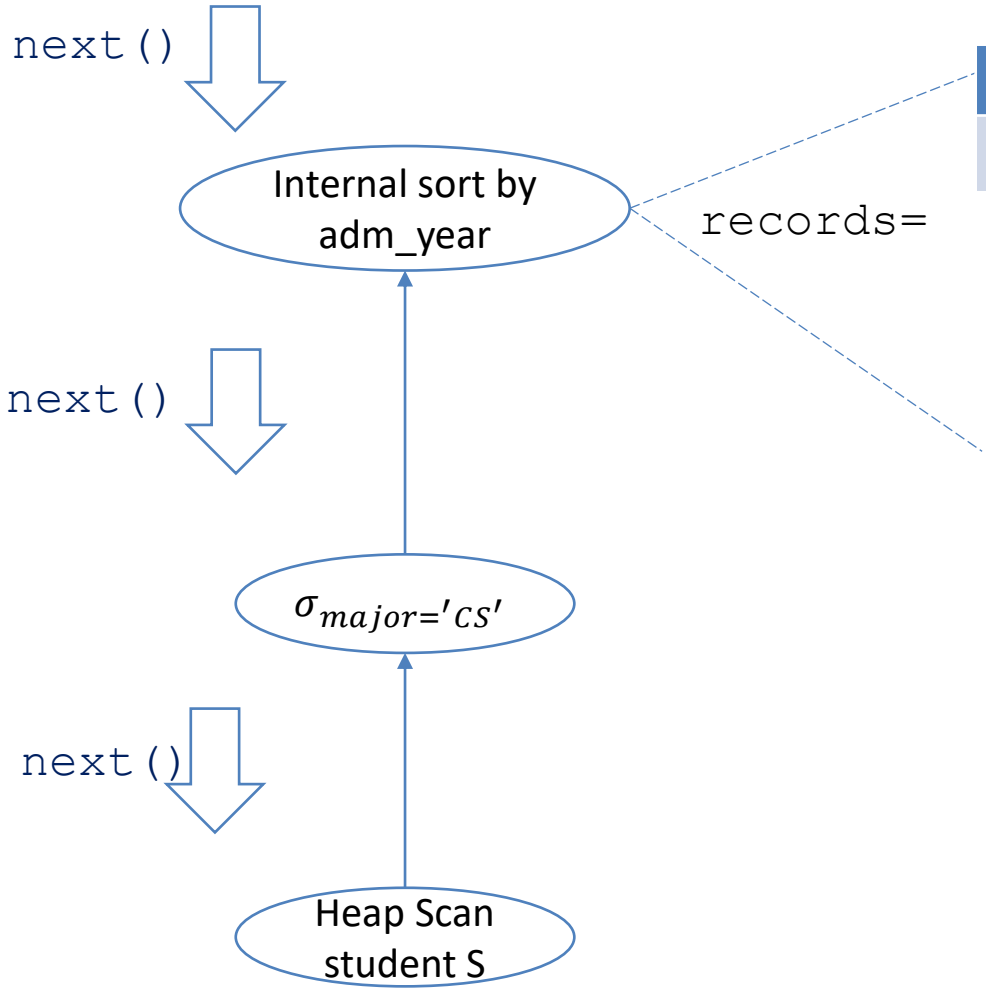
sid	name	login	major	adm_year
100	Alice	alicer34	CS	2021
101	Bob	bob5	CE	2020
102	Charlie	charlie7	CS	2021
103	David	davel	CS	2020

Example: putting it together



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

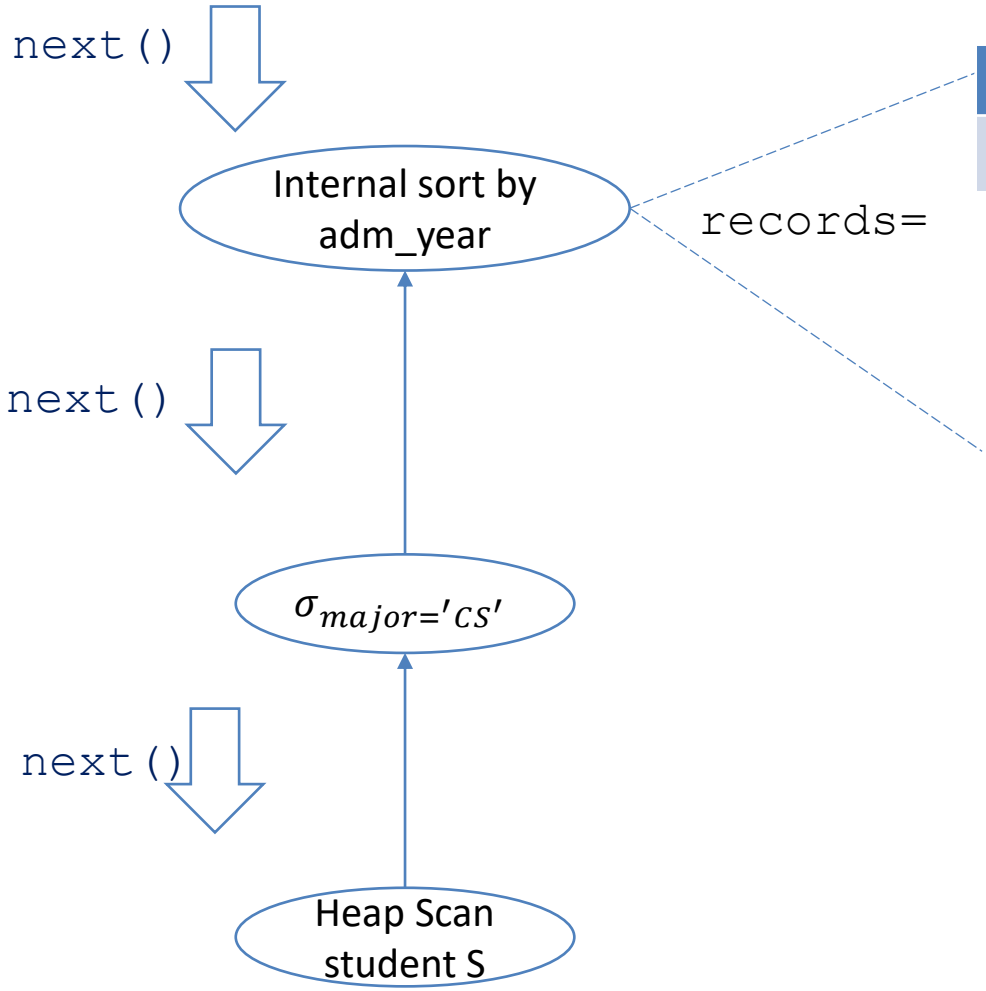
Example: putting it together



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

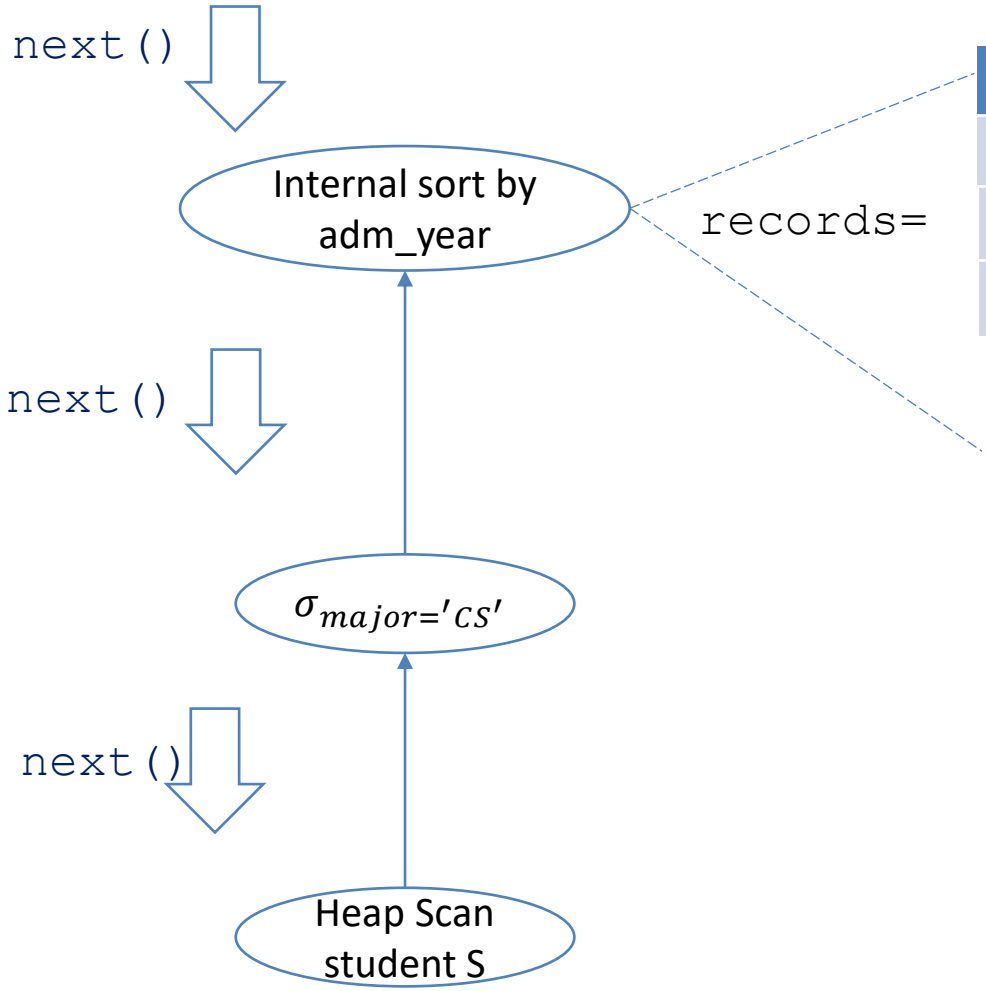
Example: putting it together



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

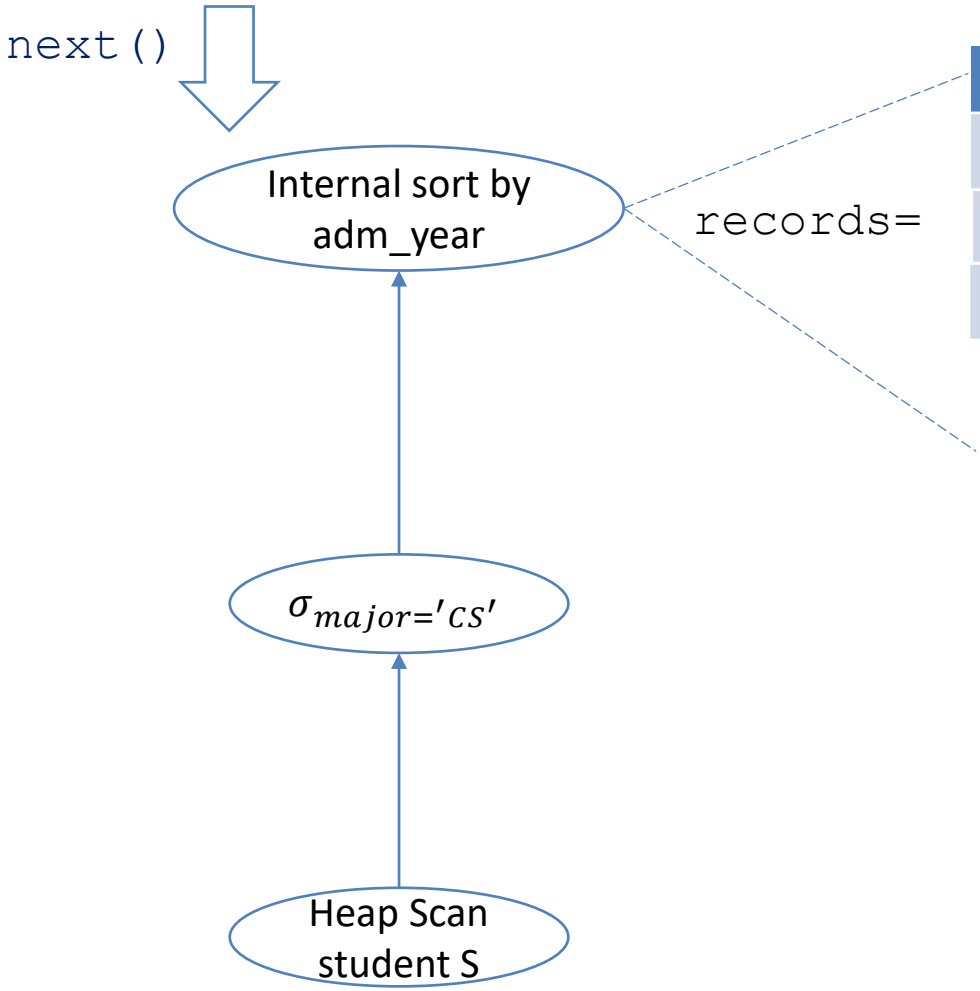
Example: putting it together



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

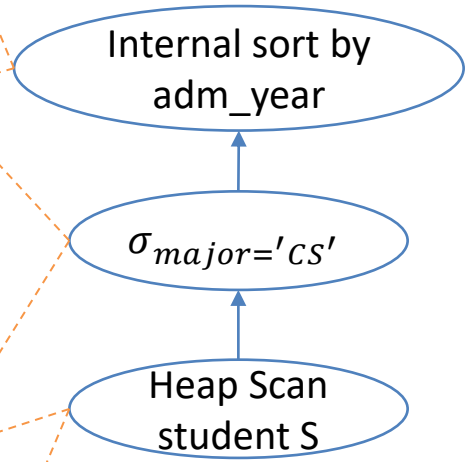
Materialization model

- 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;
```



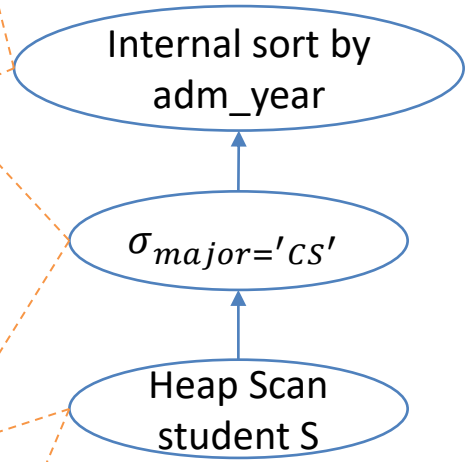
Vectorization model

- 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

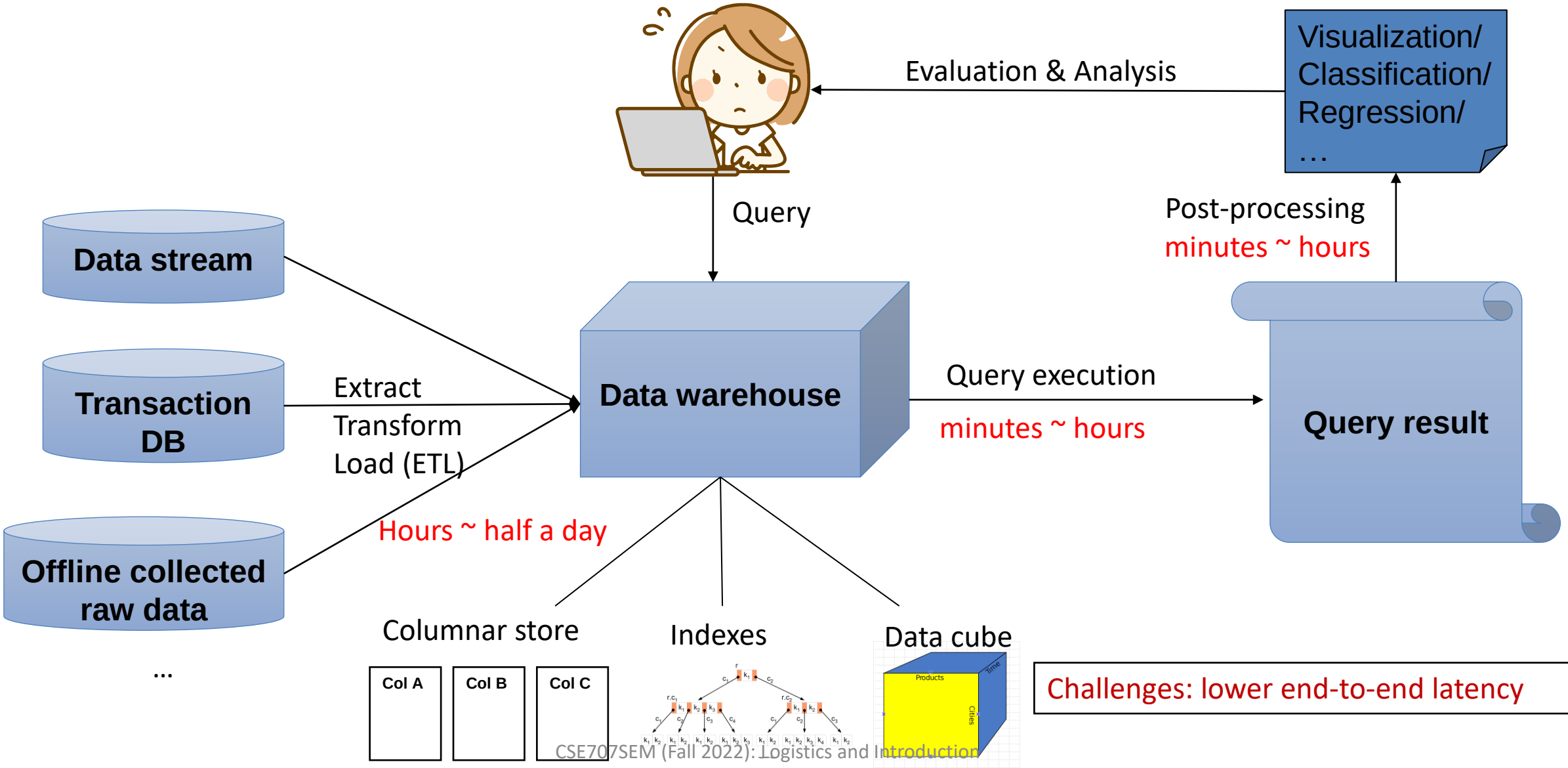
```
out = []
while c_out = child.Next():
   out.extend(c_out)
sort(out)
return out
```

```
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
```

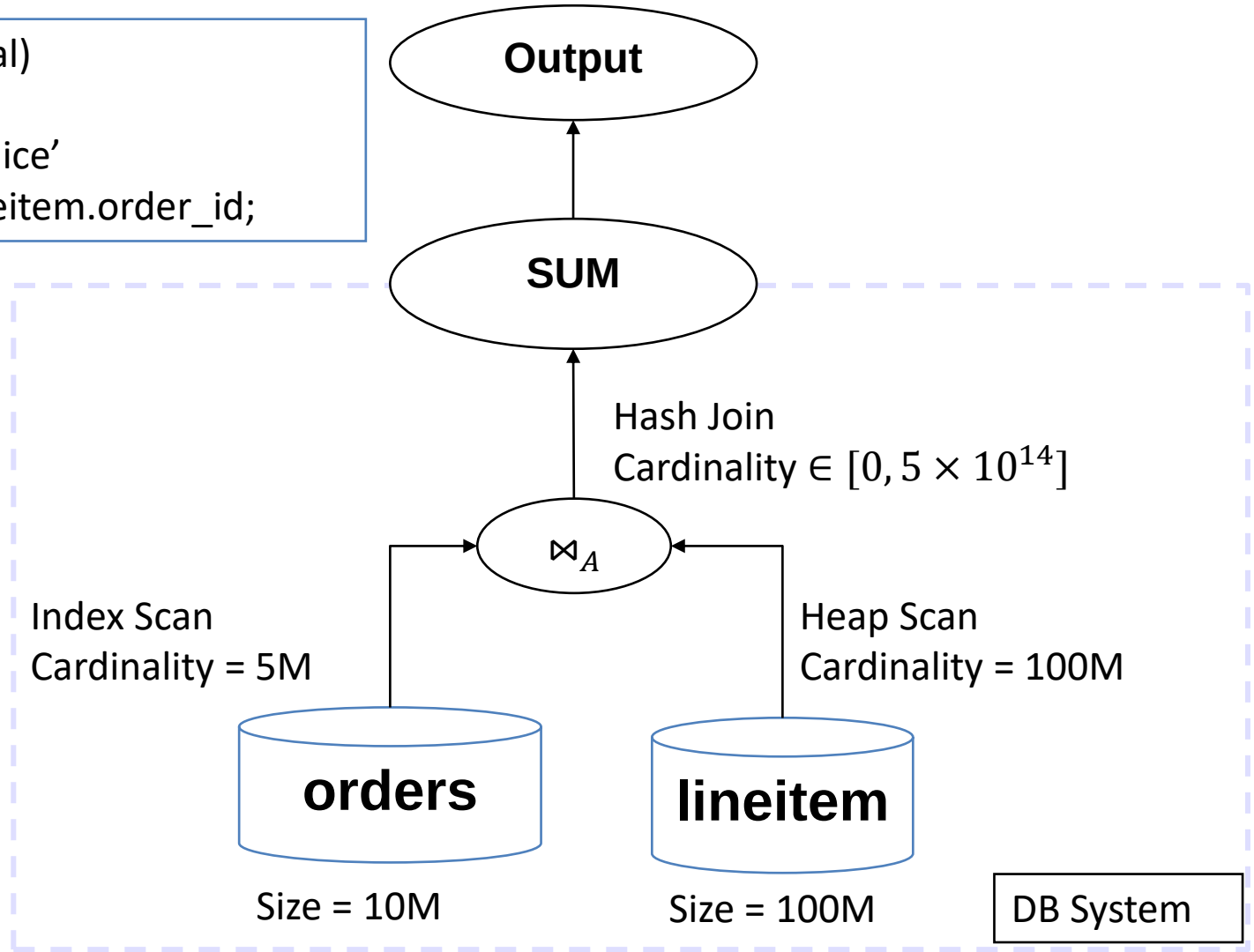


Motivation for Approximate Query Processing



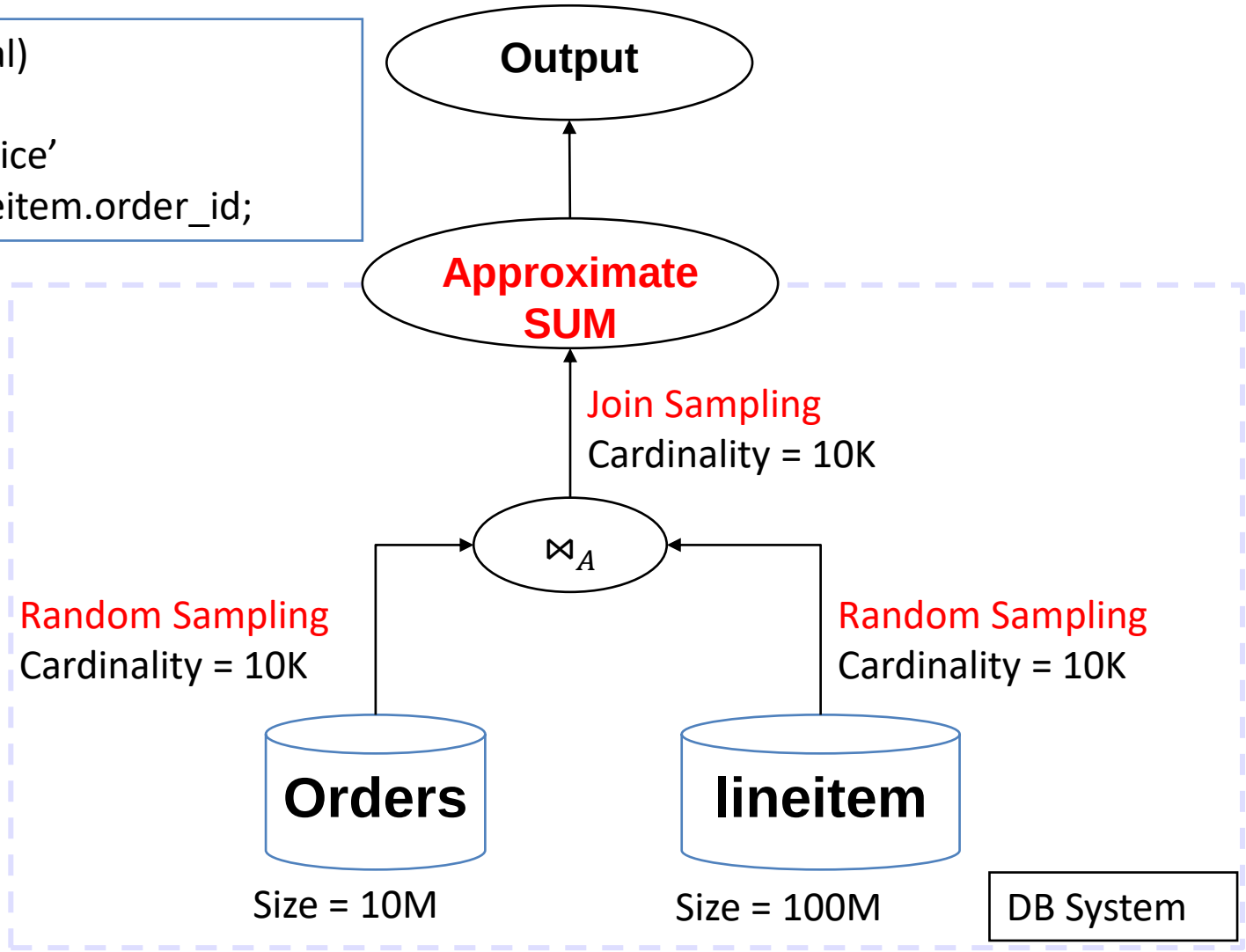
Approximate Query Processing

```
SELECT SUM(lineitem.subtotal)
FROM orders, lineitem
WHERE orders.customer = 'Alice'
    AND orders.order_id = lineitem.order_id;
```

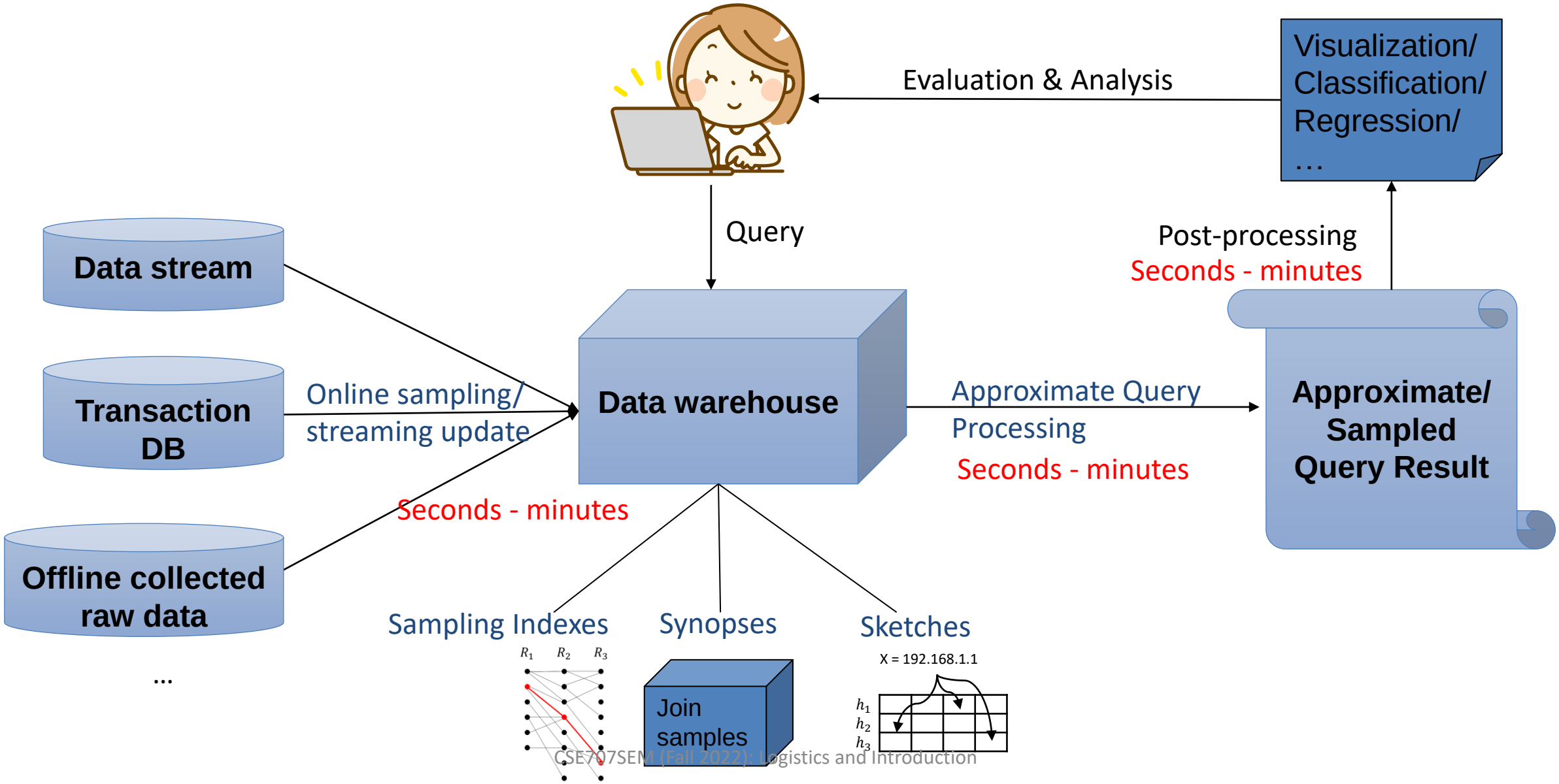


Random Sampling to Reduce Intermediate Sizes

```
SELECT SUM(lineitem.subtotal)
FROM orders, lineitem
WHERE orders.customer = 'Alice'
    AND orders.order_id = lineitem.order_id;
```



Approximate Query Processing as a cheaper alternative



Presentation Sign up