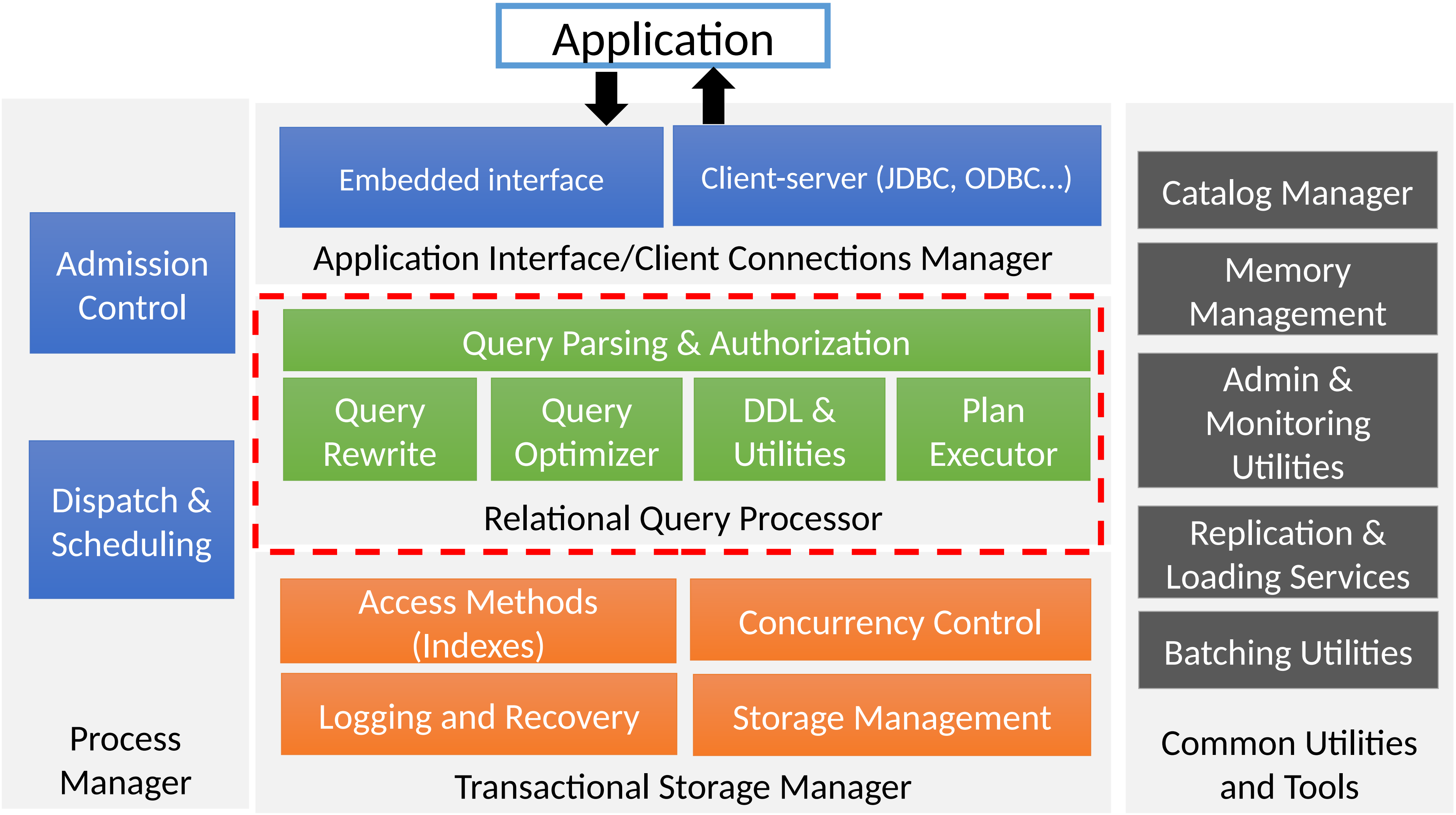
CSE 541: Database Systems I

Query Processing

Slides Credit: Tianzheng Wang

DBMS Components



SQL Query Text ➔ Logical Plan

Table schemas:

```
Sailors(sid: integer, sname: string, rating: integer, age: real)
Reserves(sid: integer, bid: integer, day: dates, rname: string)
```

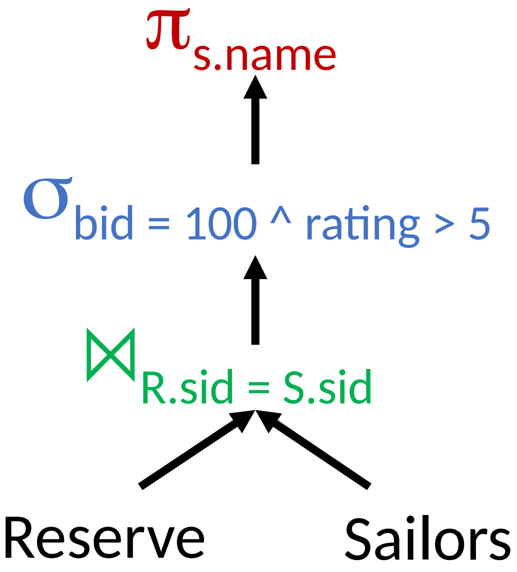
SQL query (declarative):

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

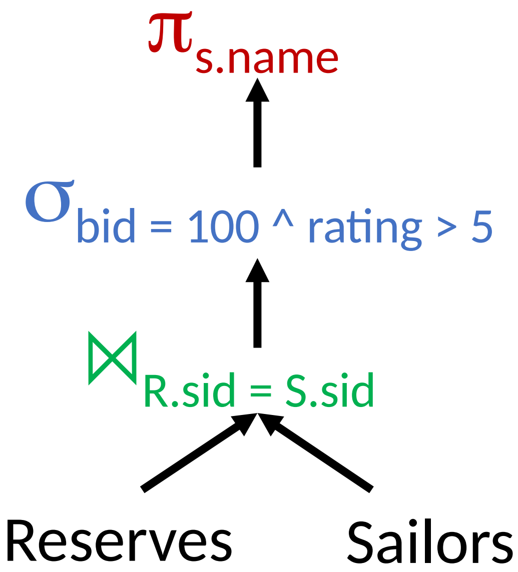
SQL parser

$\pi_{s.name}(\sigma_{bid = 100 \wedge rating > 5}(\text{Reserves} \bowtie_{R.sid=S.sid} \text{Sailors}))$

Equivalent logical query plan
(relational algebra tree):



Relational Operators and Query Plans



Edges: "flow" of tuples

Vertices: relational algebra operators

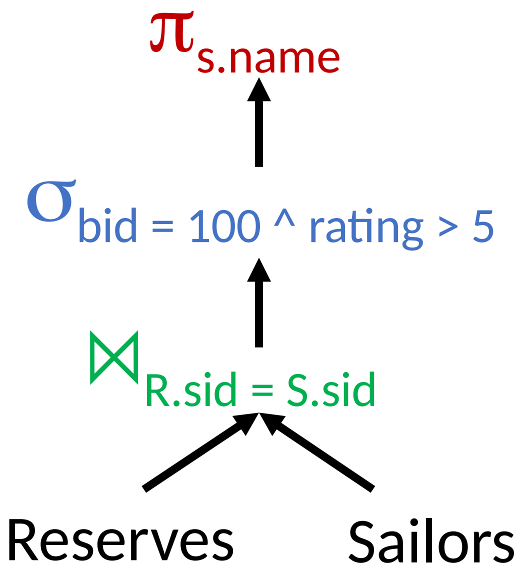
- Input/output: relation

- Aka "data-flow" graph, also used in other systems

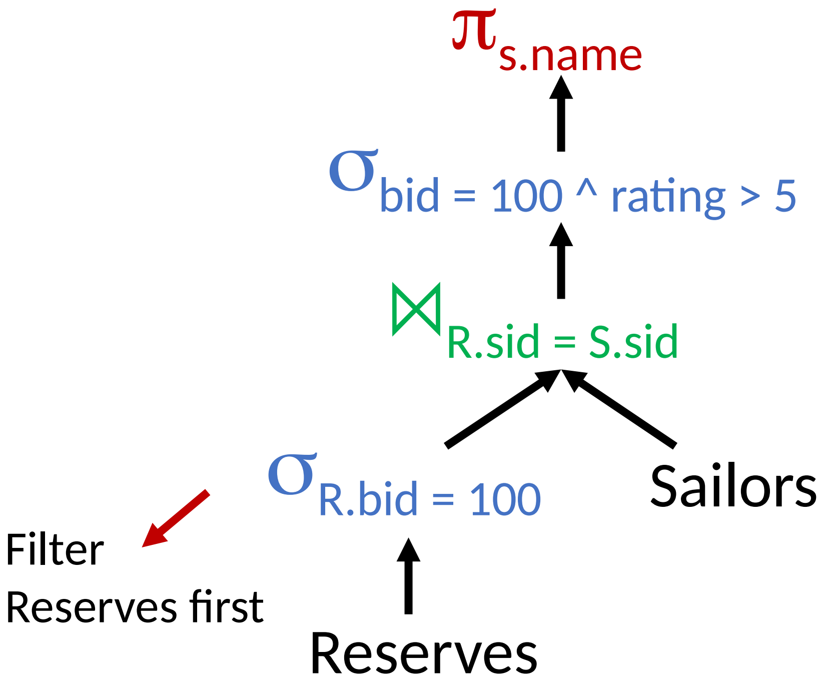
- Query optimizer determines the implementation to use for each operator
- Query executor runs the relational operators

Logical Plan ➔ Optimized Logical Plan

Logical query plan:



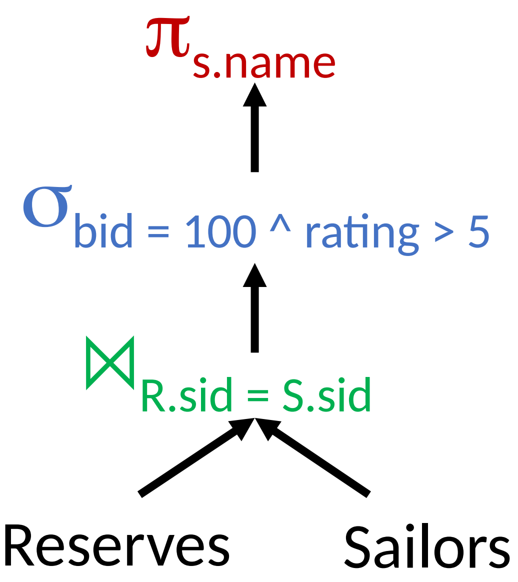
Optimized logical query plan:



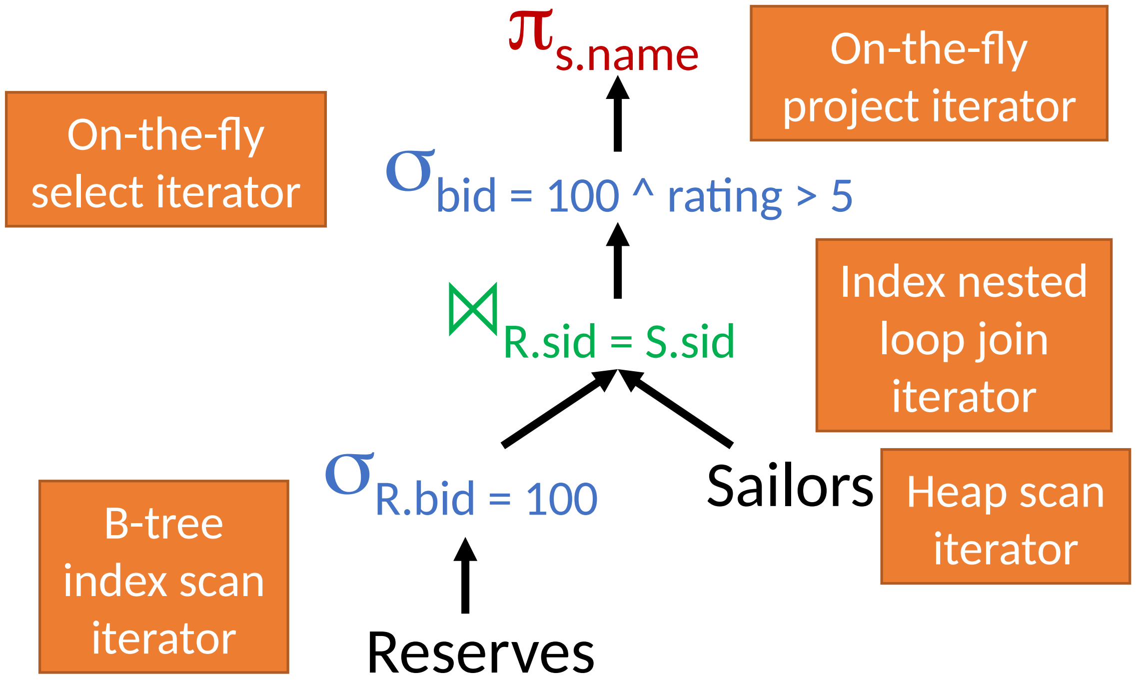
- There could be many different ways to optimize a plan
- Goal of query optimization
  - Ideal: come up with the best plan (lowest cost)
  - Reality: avoid the worst plans

Optimized Logical Plan ➔ Physical Plan

Logical query plan:



(Optimized) physical query plan:



- Each relational operator is handled separately and implemented as an iterator instance that is a subclass of a base iterator class

Iterator Interface (Volcano Model)

```
abstract class iterator {
  void open(args);
  tuple next();
  void close();
};
```

open:

- Initialize the iterator
- Allocate buffers for input/output
- Pass in arguments (e.g., selection condition)

Pull based computation:

- A single thread starts query execution by calling init/next function of the top-most node which recursively calls init/next of lower level nodes (“children”)

Encapsulation:

- No need to know the exact subclass implementation – they all follow the same interface
  - E.g., using C++ abstract class and virtual functions

Iterator Interface (Volcano Model)

Internal state:

- Iterator may maintain states not shared with other iterators
  - E.g., hash tables for join, sorted files… (these can be large)
  - But operator results are not stored permanently – they are streamed through the plan's call stack (following the 'edges')

Iterator behavior:

- On-the-fly (streaming)
  - E.g., return a tuple on each next() call
  - Small, constant amount of work per call
- Blocking (batch)
  - E.g., keep calling next() of child, then return the first tuple
  - No output produced until the entire input is consumed

Example: Heap Scan Iterator

```
class heap_scan_iterator : public iterator {
  void open(relation) {
    heap = open file for relation;
    cur_page = heap.first_page();
    cur_slot = cur_page.first_slot();
  }
  RID next() {
    if (cur_page.id == invalid) return EOF;
    rid = [cur_page.id, cur_slot.id]

    cur_slot = cur_page.next_slot();
    if (cur_slot.id == invalid) {
      cur_page = cur_page.next();
      if (cur_page.id != invalid)
        cur_slot = cur_page.first_slot();
    }
    return rid;
  }
  void close() { heap.close(); }
};
```

Already at the leaf level of the plan, no children, i.e., no `child.open()`

No more records to look at in this file

The result to be returned

Advance the "cursor" for the next iteration

Advance to the next page if needed

Example: On-the-Fly Select Iterator

```
class select_iterator : public iterator {
  void open(predicate) {
    child.open();
    pred = predicate;
    current = nullptr;
  }

  void close() {
    child.close();
  }

  tuple next() {
    while (current != EOF && !pred(current)) {
      current = child.next();
    }
    return current;
  }
};
```

Predicate for the operator (e.g., Student GPA > 3)

The output tuple to be returned

"End of file" - no more candidate tuples

Check whether the current candidate tuple matches the provided predicate

Ask the lower level node for input

Local states:

- `pred`
- `current`

Two-Pass Sort Iterator

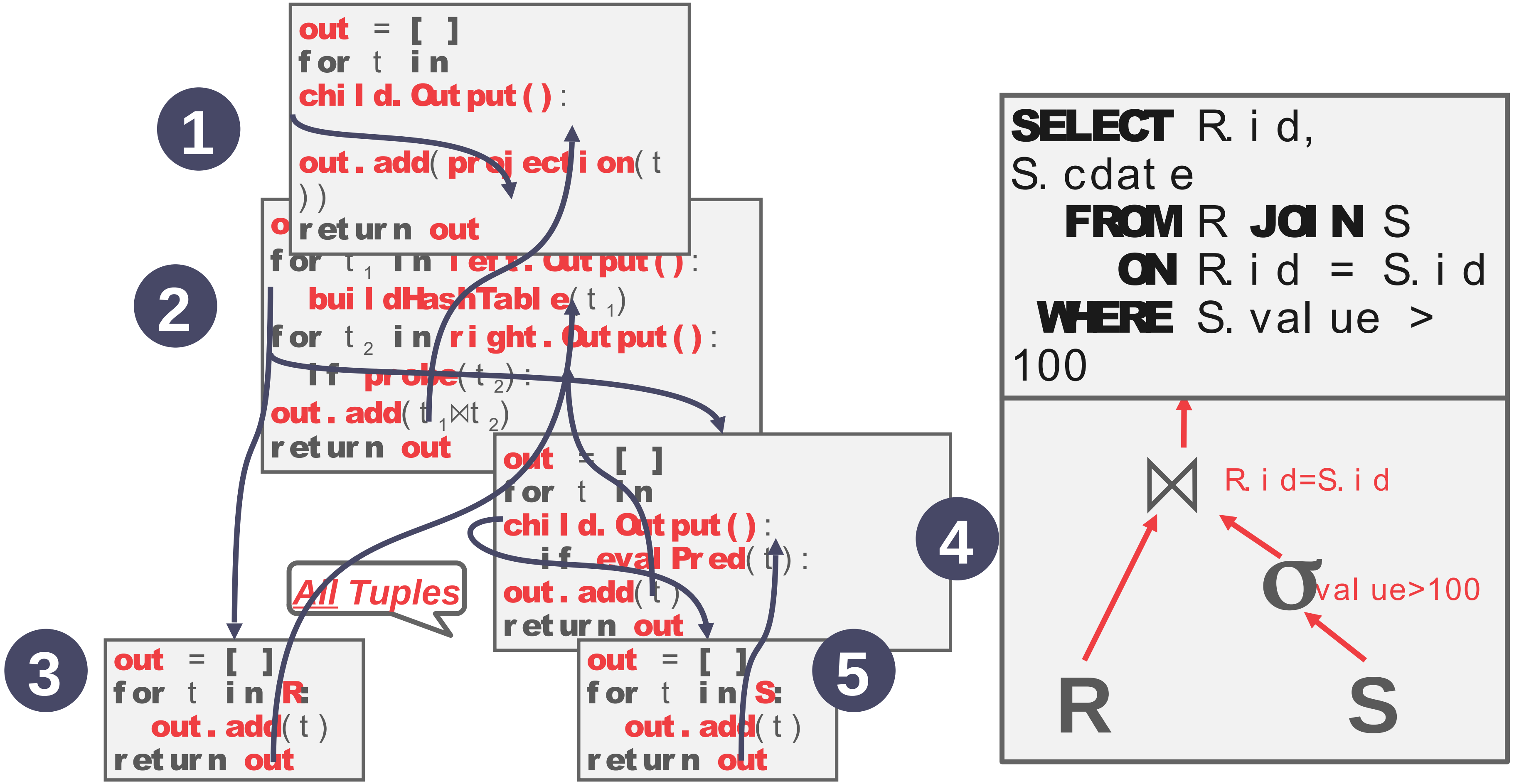
```
class 2pass_sort_iterator : public iterator {
  void open(keys) {
    child.open();
    repeatedly calling child.next() to fetch tuples and
    generated sorted runs in disk, until child hits EOF
    open each sorted run file, load it into input buffer
  }
  void next() {
    output = min tuple across all buffers
    if min tuple is the last one in its buffer, fetch the
next page
    return output or EOF if there are no more tuples
  }
  void close() {
    deallocate all sorted run files
    child.close();
  }
};
```

Materialization Model

- Each operator processes its input all at once and then emits its output all at once.
  - The operator "materializes" its output as a single result.
  - The DBMS can push down hints into to avoid scanning too many tuples.
  - Can send either a materialized row or a single column.

- The output can be either whole tuples (NSM) or subsets of columns (DSM)

Materialization Model



Materialization Model

- Better for OLTP workloads because queries only access a small number of tuples at a time.
  - Lower execution / coordination overhead.
  - Fewer function calls.

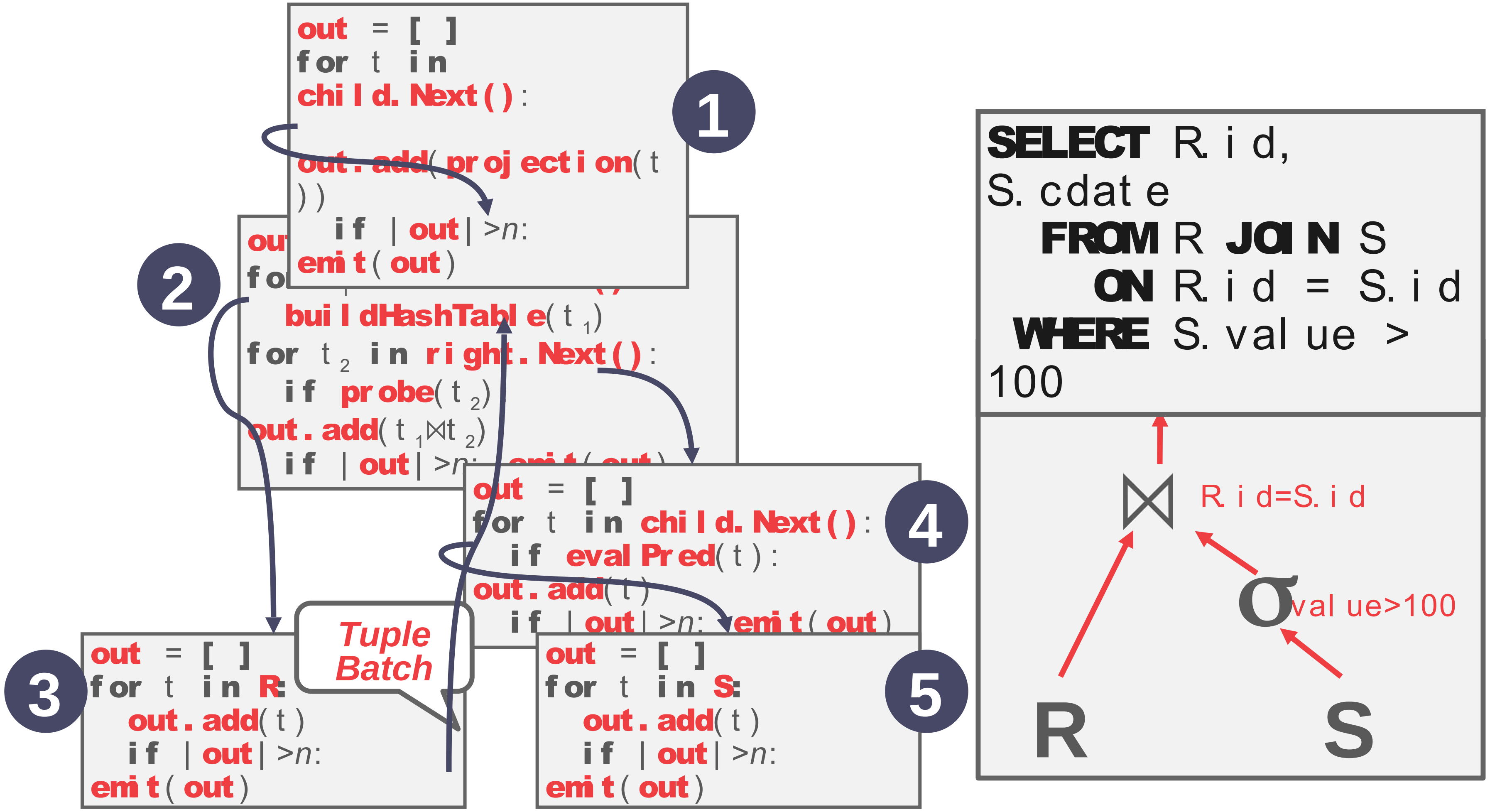
- Not good for OLAP queries with large intermediate results.

Vectorization Model

- Like the Iterator Model where each operator implements a **Next** function in this model.

- Each operator emits a **batch** of tuples instead of a single tuple.
  - The operator's internal loop processes multiple tuples at a time.
  - The size of the batch can vary based on hardware or query properties.

Vectorization Model



Vectorization Model

- Ideal for OLAP queries because it greatly reduces the number of invocations per operator.
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.

presto vectorwise snowflake HYRISE DuckDB noise page

Microsoft SQL Server IBM DB2 ORACLE amazon REDSHIFT

Plan Processing Direction

- **Approach #1: Top-to-Bottom**
  - Start with the root and "pull" data up from its children.
  - Tuples are always passed with function calls.

- **Approach #2: Bottom-to-Top**
  - Start with leaf nodes and push data to their parents.
  - Allows for tighter control of caches/registers in pipelines.

Summary

- SQL queries get translated into query plans
  - Represented by relational algebra trees with operators
  - A graph, aka “data-flow” model
- Iterator interface (Volcano Model)
  - Each operator implements an abstract iterator interface with open, next, close functions
  - Pull based model: query gets evaluated by a single thread invoking the open/next functions which recursively calls into the lower-level children’s open/next functions
  - Can be streaming or blocking
  - Example iterator implementations (select, heap scan)
- Catalogs record information needed by query evaluation and optimization