Tutorial 4: Query Execution

Implementation of Databases (DBS2)

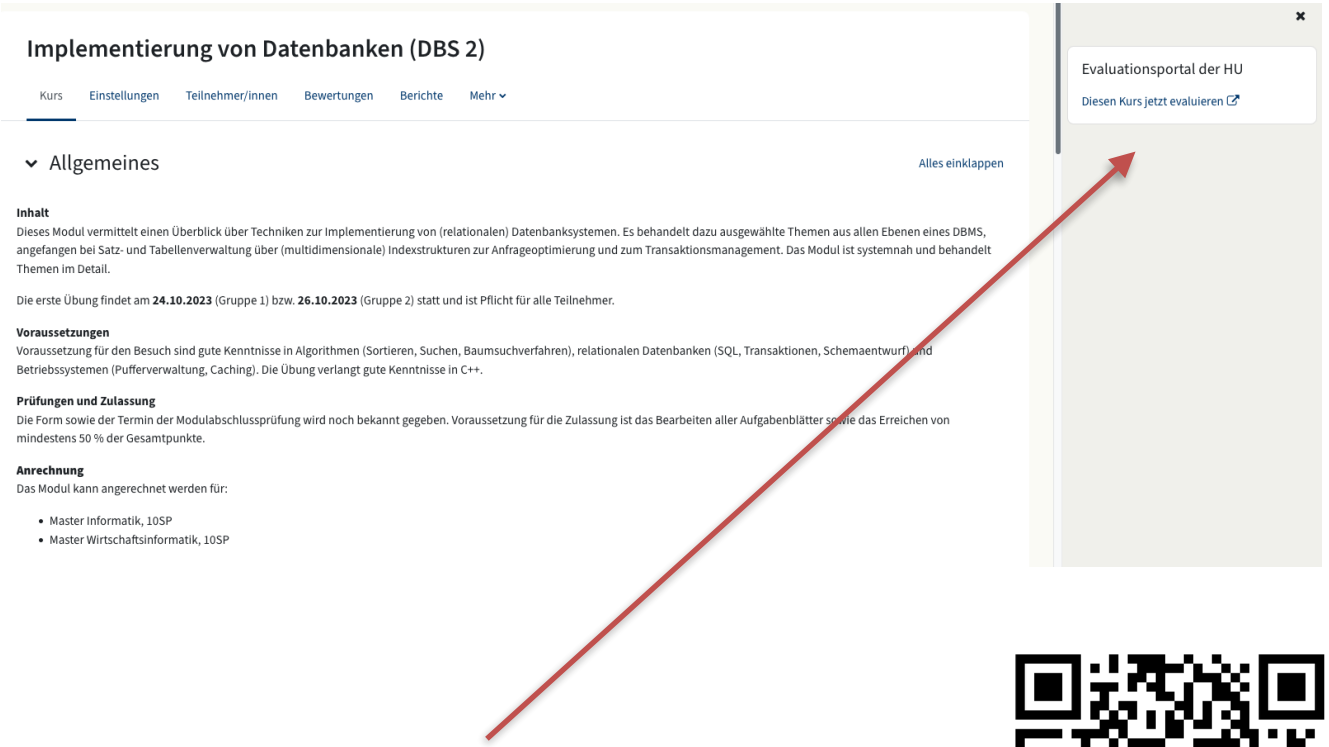
Arik Ermshaus

Tutorial appointments

Week	Topic
16.10 - 20.10	-
23.10 - 27.10	Organisation, Exercise Sheet 1
30.10 - 03.11	Q&A
06.11 - 10.11	Q&A
13.11 - 17.11	Exercise Sheet 2
20.11 - 24.11	Q&A
27.11 - 01.12	Q&A
04.12 - 08.12	Exercise Sheet 3
11.12 - 15.12	Q&A
18.12 - 22.12	Q&A
25.12 - 29.12	-
01.01 - 05.01	-
08.01 - 12.01	**Exercise Sheet 4**
15.01 - 19.01	Q&A
22.01 - 26.01	Q&A
29.01 - 02.02	Exercise Sheet 5
05.02 - 09.02	Q&A
12.02 - 16.02	Exam preparation

Disclaimer: Timetable is provisional, and will (probably) change!

Lehrevaluation



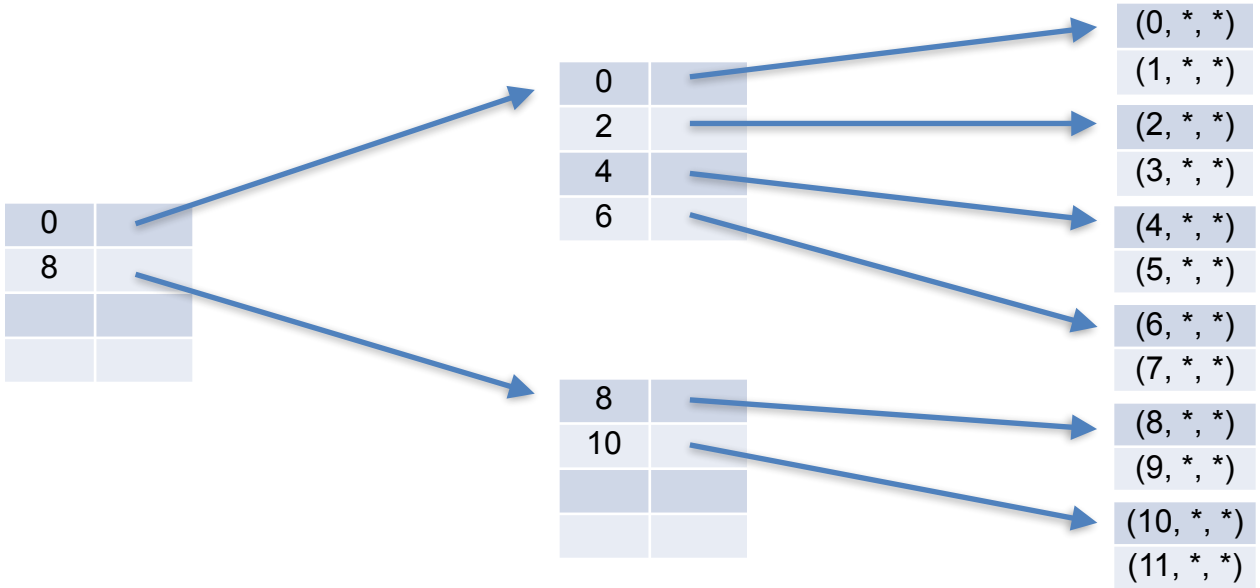
Please take 5 minutes to fill out the Lehrevaluation on Moodle:
https://moodle.hu-berlin.de/course/view.php?id=122985

Table of Contents

Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

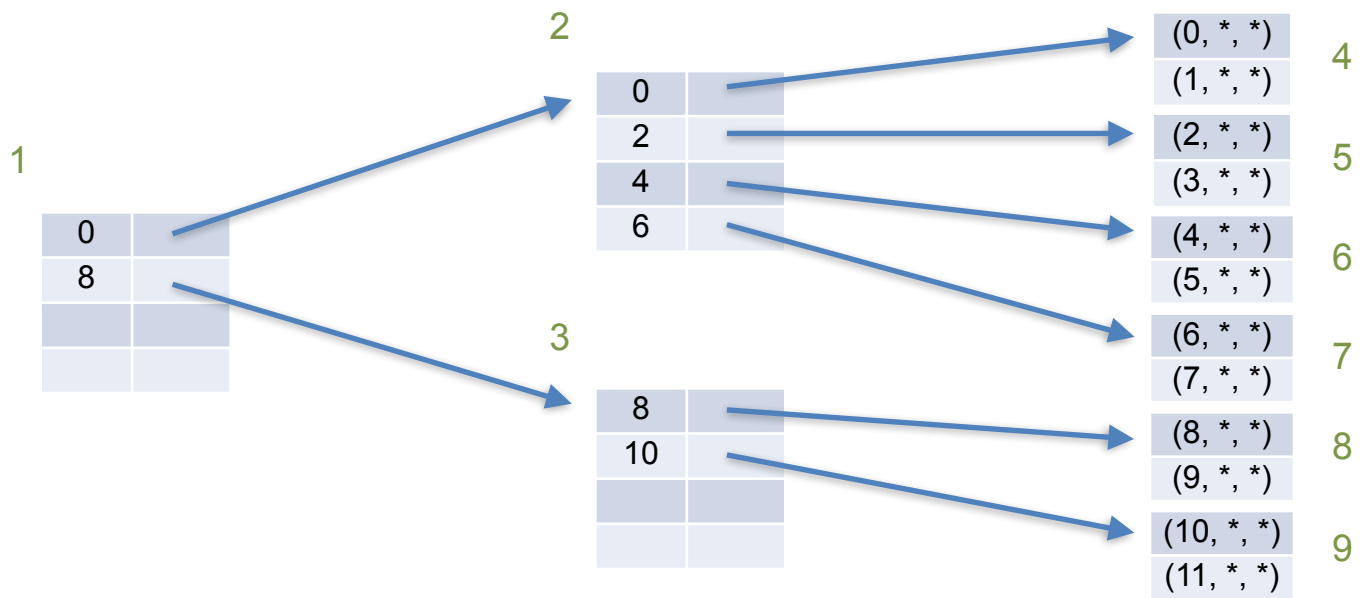
(a) Index given data file with two-level sparse primary index *I* on A.



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

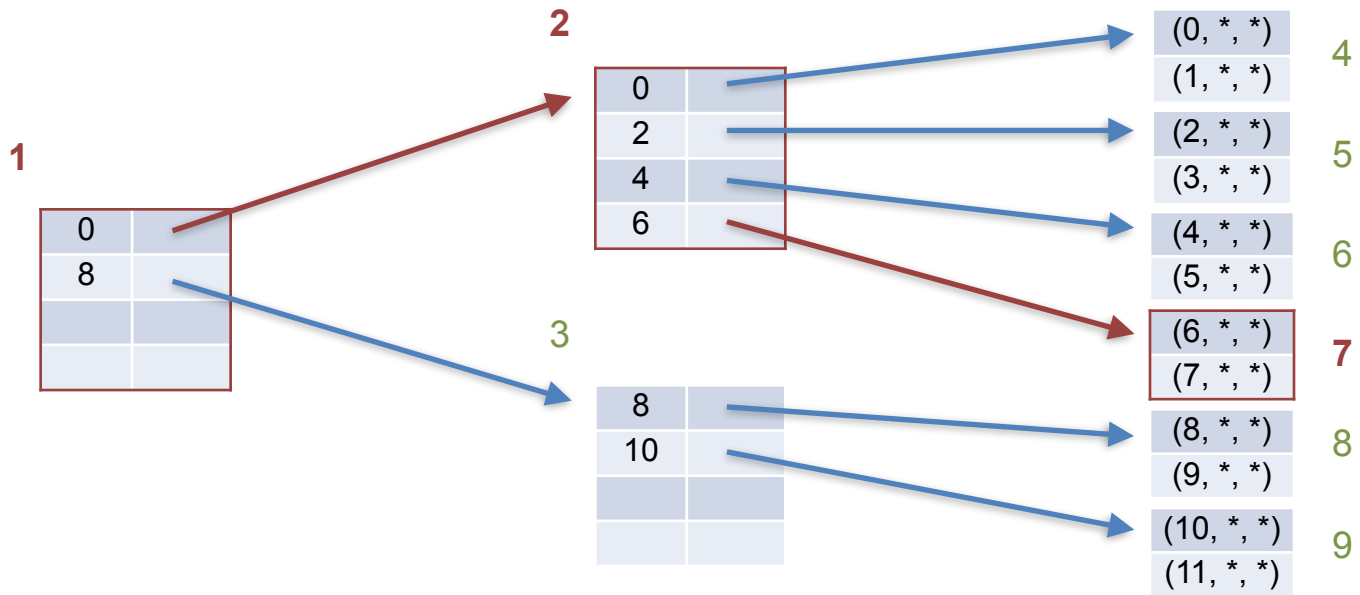
(b) Annotate index structure *I* with unique identifier.



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

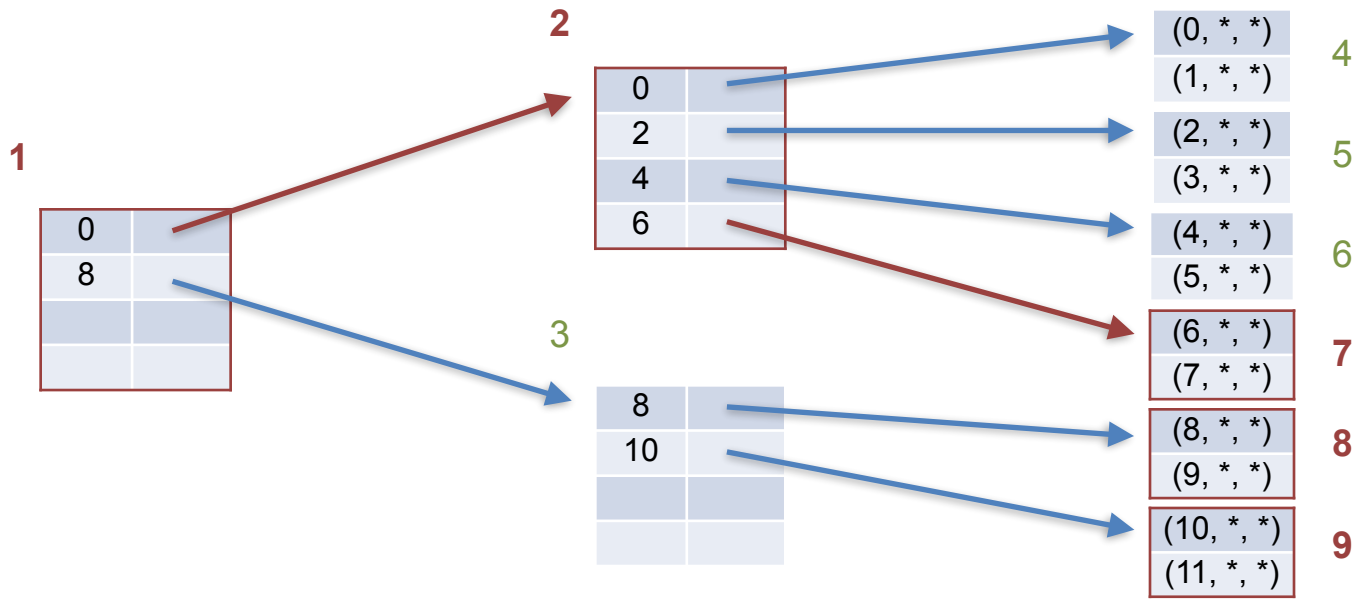
(b) Specify blocks that must be read for query:
SELECT * FROM R WHERE A = 6



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

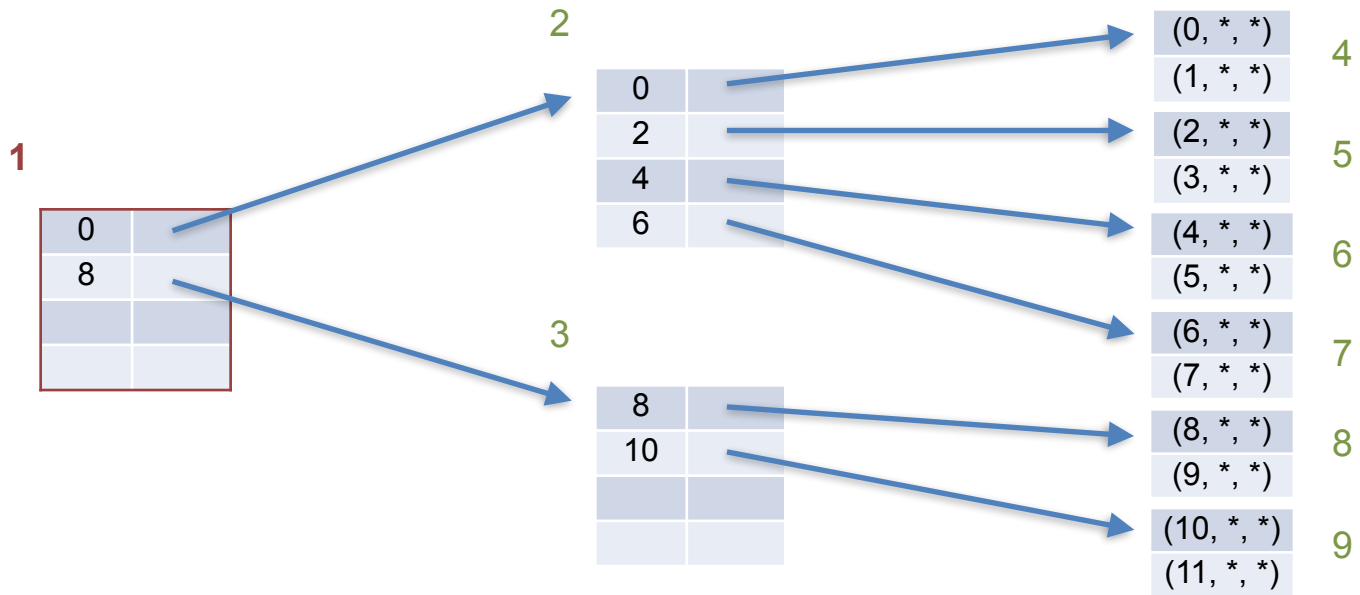
(b) Specify blocks that must be read for query:
SELECT * FROM R WHERE A > 7



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

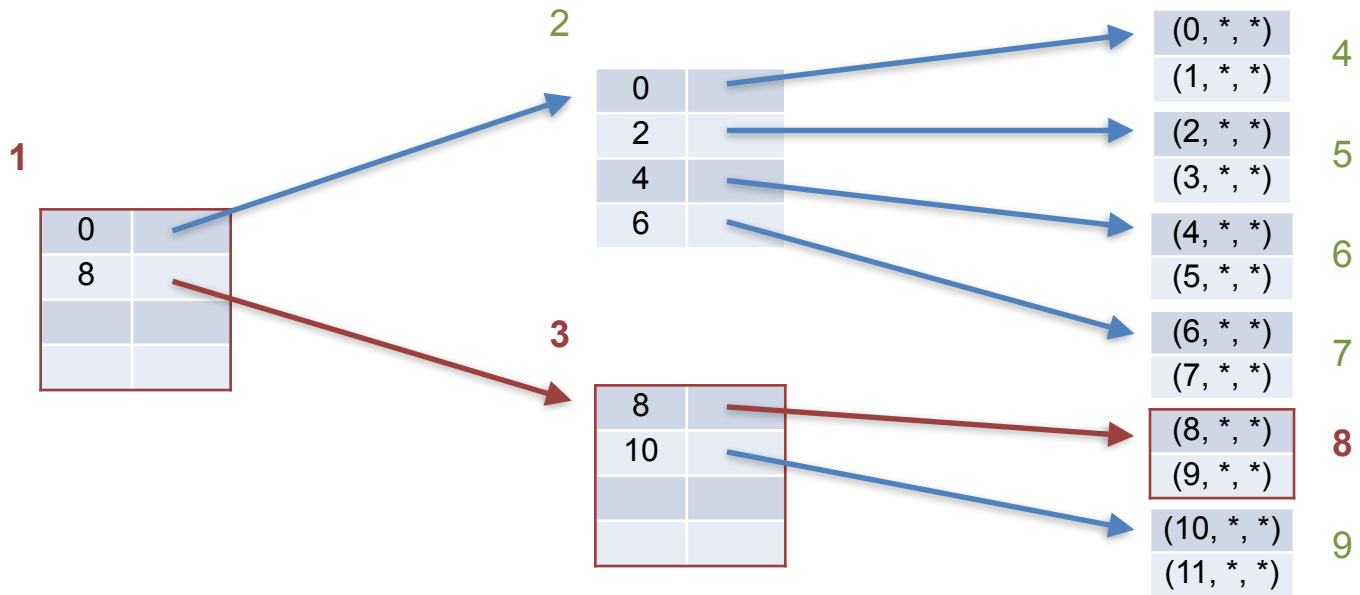
(b) Specify blocks that must be read for query:
SELECT COUNT(*) FROM R WHERE A = 8



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

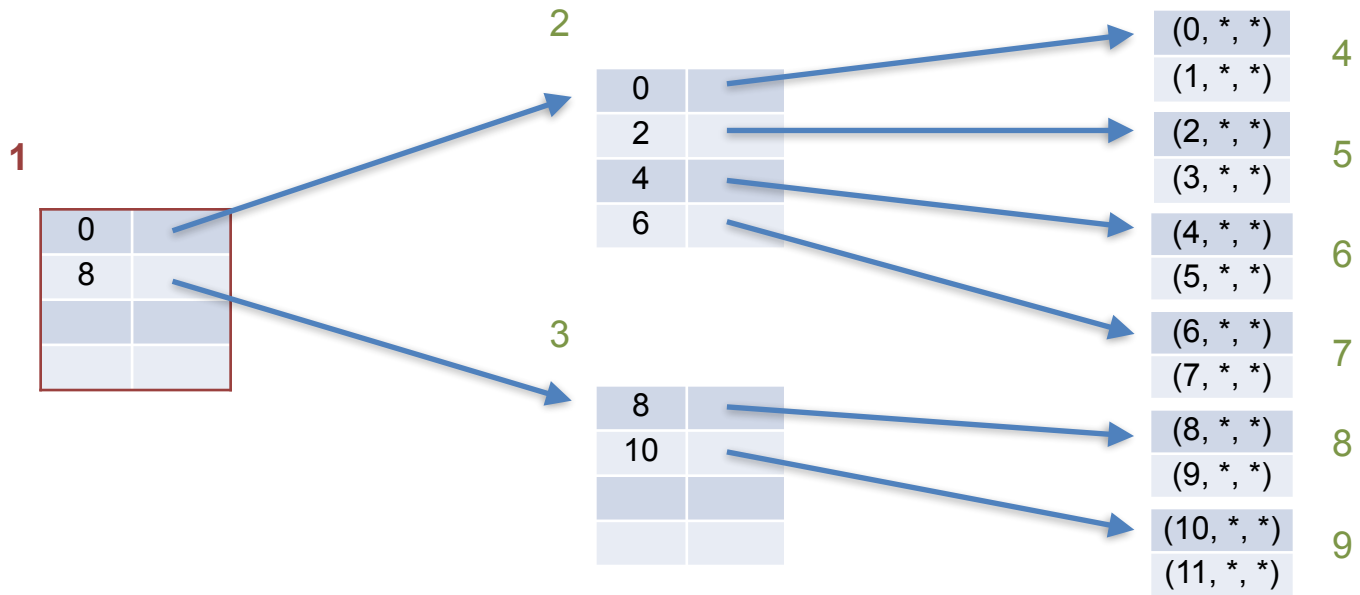
(b) Specify blocks that must be read for query:
SELECT COUNT(*) FROM R WHERE A = 9



Task 1: Hierarchical Index-Sequential Files

- Relation R(A, B, C) comprises 12 tuples
- Stored sorted (by primary key A) in ISF
- Block can store 2 tuples or 4 key-pointer Paris

(b) Specify blocks that must be read for query:
SELECT MIN(A) FROM R



Task 2: Hash Files

- Relation S(A, B, C) comprises 16384 tuples
- S stored in hash file with blocks of 128 records
- Access: uniform hash function h(B), hash table *H* (64 entries)

(a) What is amount of records per bucket in hash file?

- $B_{records} = \frac{\#_{records}}{\#_H} = \frac{2^{14}}{2^6} = 2^8 = 256$

Task 2: Hash Files

- Relation S(A, B, C) comprises 16384 tuples
- S stored in hash file with blocks of 128 records
- Access: uniform hash function h(B), hash table *H* (64 entries)

(b) How many blocks have to be loaded to retrieve records for single value of B. Assume an unsuccessful search.

- In total, we have $\frac{2^{14}}{2^7} = 2^7 = 128$ blocks
- Blocks are equally distributed over 64 buckets
- Hence, each bucket has exactly 2 blocks
- Both blocks have to be loaded in order to search for the value

Task 2: Hash Files

- Relation S(A, B, C) comprises 16384 tuples
- S stored in hash file with blocks of 128 records
- Access: uniform hash function h(B), hash table *H* (64 entries)

(c) Why is hash file bad choice of data structure for range query?

- Lack of order: records are not sorted, we may have to scan many more than we return
- Much random IO: buckets contain blocks at random positions in file
- Skew and degradation: also leads to much more records that need to be checked than actually returned

Task 3: B+ tree

Complete the provided B+ tree structure with node insert and splitting functionality.

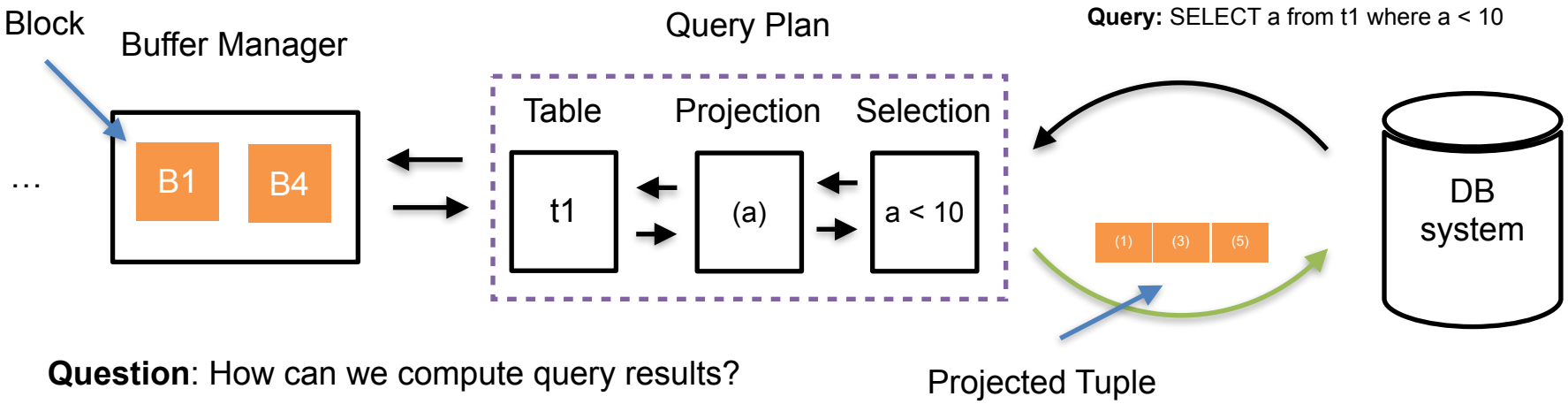
```cpp
std::optional<std::pair<std::shared_ptr<BPTreeNode>, int>> BPTreeNode::insert_record(int attribute,
std::string const& record_id)
{
    assert(is_leaf());

    std::vector<int> values = get_values();
    std::vector<std::string> children_ids = get_children_ids();

    // find correct position to insert new attribute
    auto it = std::lower_bound(values.begin(), values.end(), attribute);

    // handle duplicates correctly
    if (it != values.end() && *it == attribute)
        throw std::invalid_argument("Found duplicates in index block: " + block_id);

    // insert attribute and children_id at correct position
    size_t insert_pos = it - values.begin();
    values.insert(it, attribute);
    children_ids.insert(children_ids.begin() + insert_pos, record_id);

    // check if block is over-full
    if (values.size() <= BPTreeNode::MAX_VALUES)
    {
        assert(change_values(values) && change_children_ids(children_ids));
        return std::nullopt;
    }

    // split node
    std::shared_ptr<BPTreeNode> new_node = BPTreeNode::create_node(
            buffer_manager,buffer_manager->create_new_block(),get_parent_id(), is_leaf()
    );

    std::vector<int> new_values = new_node->get_values();
    std::vector<std::string> new_children_ids = new_node->get_children_ids();

    // split values and children_ids
    size_t middle_index = values.size() / 2;
    auto middle = values.begin() + middle_index;

    // distribute values and children_ids
    new_values.assign(middle, values.end());
    new_children_ids.assign(children_ids.begin() + (middle - values.begin()), children_ids.end());
    values.erase(middle, values.end());
    children_ids.erase(children_ids.begin() + (middle - values.begin()), children_ids.end());

    // add new leaf chain pointer
    children_ids.push_back(new_node->get_node_id());

    int median = new_values.at(0);

    assert(change_values(values) && change_children_ids(children_ids));
    assert(new_node->change_values(new_values) && new_node->change_children_ids(new_children_ids));

    return {{new_node, median}};
}
```

```cpp
std::optional<std::pair<std::shared_ptr<BPTreeNode>, int>> BPTreeNode::insert_value(int attribute,
std::string const& left_children_id, std::string const& right_children_id)
{
    assert(!is_leaf());

    std::vector<int> values = get_values();
    std::vector<std::string> children_ids = get_children_ids();

    int n_values = values.size();

    // find correct position to insert the new attribute
    auto it = std::lower_bound(values.begin(), values.end(), attribute);

    // insert attribute and children_id in the correct position
    size_t insert_pos = it - values.begin();
    values.insert(it, attribute);

    // insert new left children if empty
    if (children_ids.size() == 0)
        children_ids.push_back(left_children_id);

    // insert new right children at the next position
    children_ids.insert(children_ids.begin() + insert_pos + 1, right_children_id);

    // check if block is over-full
    if (values.size() <= BPTreeNode::MAX_VALUES)
    {
        assert(change_values(values) && change_children_ids(children_ids));
        return std::nullopt;
    }

    // split node
    std::shared_ptr<BPTreeNode> new_node = BPTreeNode::create_node(
            buffer_manager,buffer_manager->create_new_block(),get_parent_id(), is_leaf()
    );

    // split values and children
    size_t middle_index = values.size() / 2;
    auto middle = values.begin() + middle_index;
    int median = values.at(middle_index);

    // create new vectors for the new node
    std::vector<int> new_values(values.begin() + middle_index + 1, values.end());
    std::vector<std::string> new_children_ids(children_ids.begin() + middle_index + 1,
children_ids.end());

    // Erase the values and children_ids from the original node after the median
    values.resize(middle_index);
    // Internal nodes have one more child than values
    children_ids.resize(middle_index + 1);

    assert(change_values(values) && change_children_ids(children_ids));
    assert(new_node->change_values(new_values) && new_node->change_children_ids(new_children_ids));

    // set parent ids for new node's children
    for (std::string children_id : new_node->get_children_ids())
        assert(std::make_shared<BPTreeNode>(buffer_manager,
children_id)->change_parent_id(new_node->get_node_id()));

    return {{new_node, median}};
}
```

Table of Contents

Table of Contents

Recap: Query Execution



Question: How can we compute query results?

- Task: Execute a query plan to compute requested query results
- Solution: Use chain of data structures and relational operators
  - Problem: multiple possible implementations exist
  - Use query optimisation techniques to decide (… later)
- Challenges: efficient implementations (fast and serialisable), produce small intermediate results, limited memory

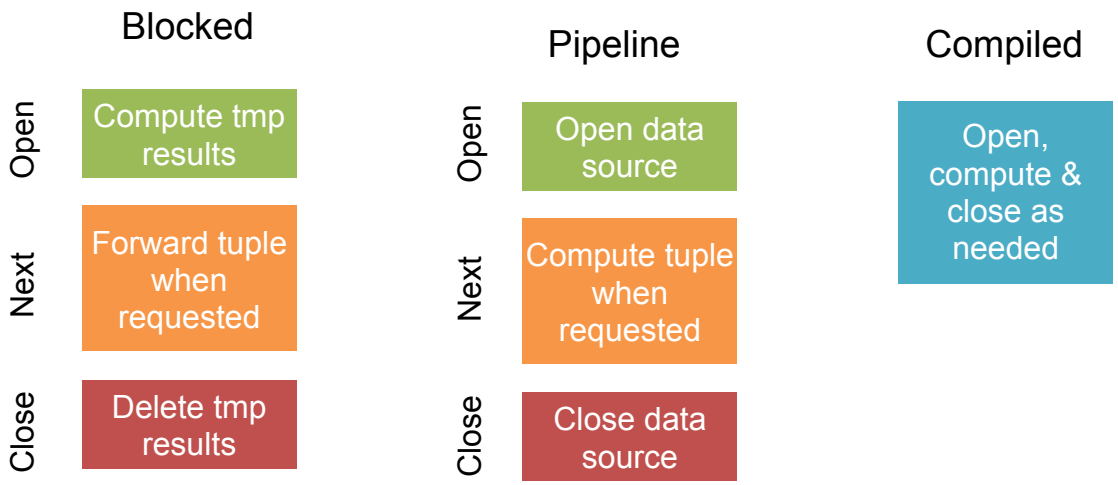
Recap: Query Execution Models

	Blocked	Pipeline	Compiled
Open	Compute tmp results	Open data source	Open, compute & close as needed
Next	Forward tuple when requested	Compute tuple when requested	
Close	Delete tmp results	Close data source	

✓ Supports complex operators	✓ Produces small intermediate results	✓ Fast, small intermediate results
✗ Produces large intermediate results	✗ Cannot implement all operators	✗ Complex implementation

- Operators call each other passing and processing tuples
  - Iterator concept: open, next & close
- Query plan generation: assemble correct sequence of operators

Task 1: Query Execution Models



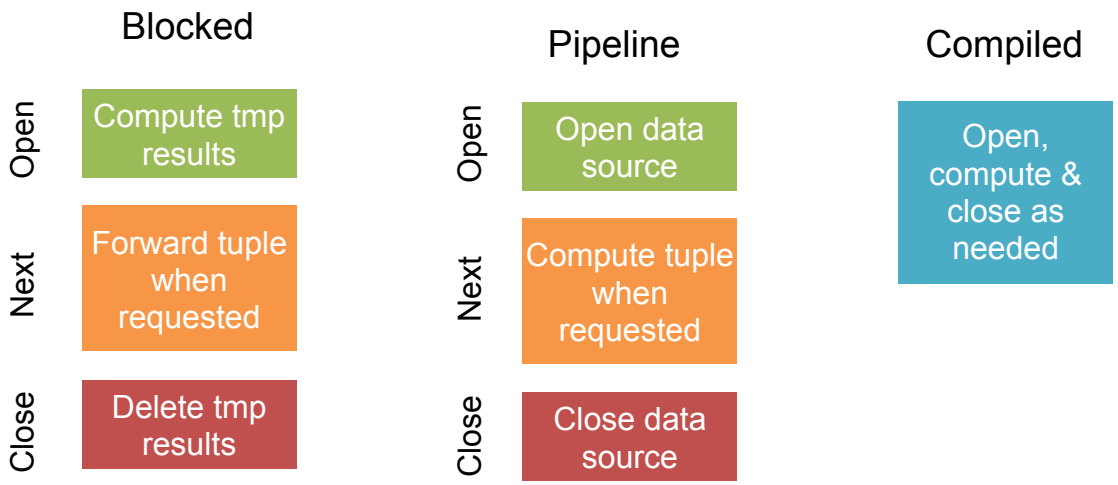
- **Question**: Which of the following functions typically has the highest runtime complexity in a "blocked" execution.

(A) Open

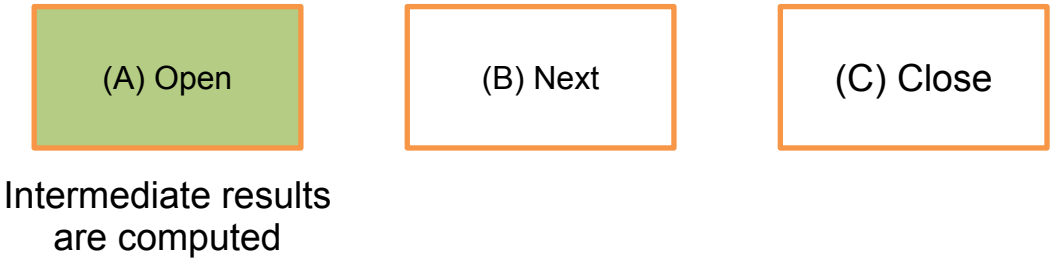
(B) Next

(C) Close

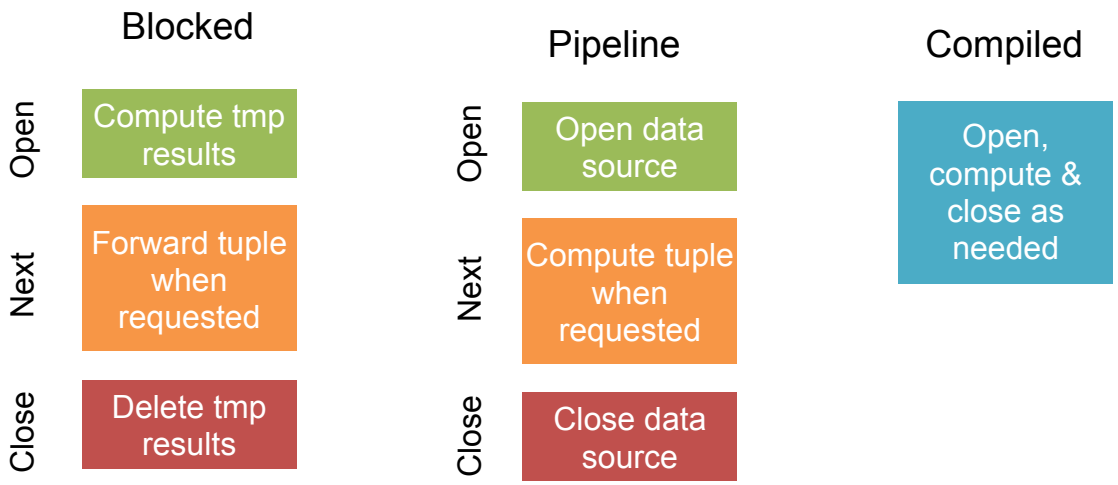
Task 1: Query Execution Models



- **Question**: Which of the following functions typically has the highest runtime complexity in a "blocked" execution.



Task 2: Query Execution Models



- **Question**: Which of the following operators can be implemented using a "pipeline".

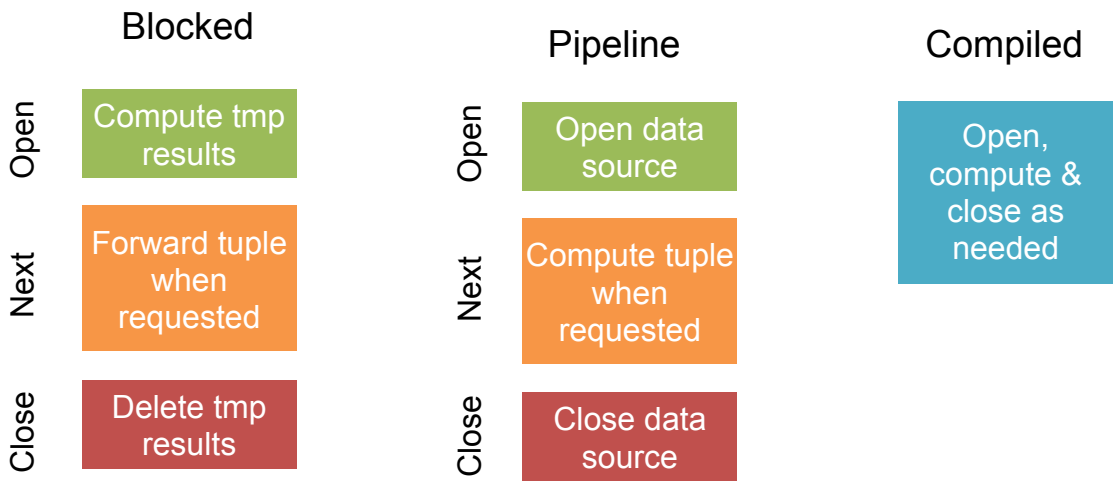
(A) INTER-SECTION

(B) SELECTION

(C) ORDER BY

(D) DISTINCT

Task 2: Query Execution Models



- **Question**: Which of the following operators can be implemented using a "pipeline".

(A) INTER-SECTION

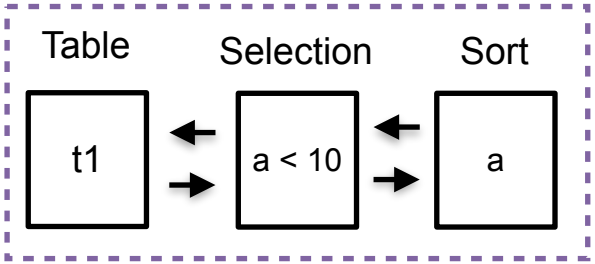
(B) SELECTION

(C) ORDER BY

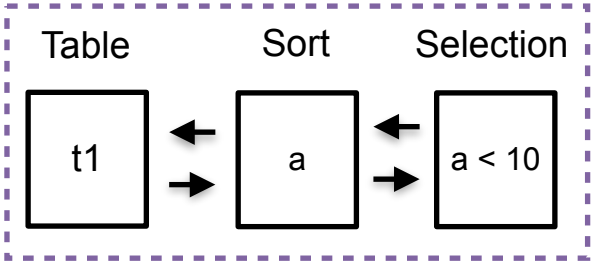
(D) DISTINCT

Task 3: Query Execution Models

Query Plan 1



Query Plan 2



Info: a is primary key of t1 in range of 0 to 99999

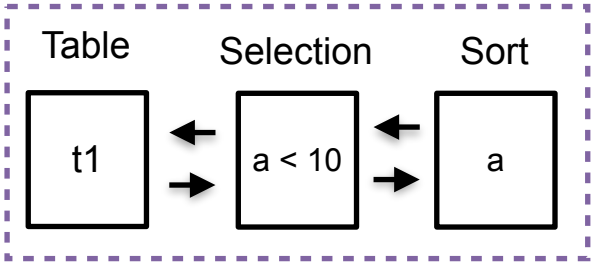
- **Question**: Which of the above query plans is computationally more efficient to compute?

(A) Plan 1

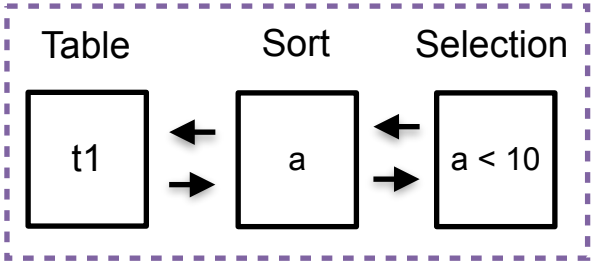
(B) Plan 2

Task 3: Query Execution Models

Query Plan 1



Query Plan 2



Info: a is primary key of t1 in range of 0 to 99999

- **Question**: Which of the above query plans is computationally more efficient to compute?

(A) Plan 1

(B) Plan 2

Selection drastically reduces intermediate results

Example: Projection Operator

Open	Next	Close
Open data source	Forward specified attributes	Close data source

Parameters: requested attributes

Open t1

(1, “Test”, true)

(2, “Test”, true)

(3, “Test”, false)

Parameters: 1st attribute

Example: Projection Operator

Open

Open data source

Next

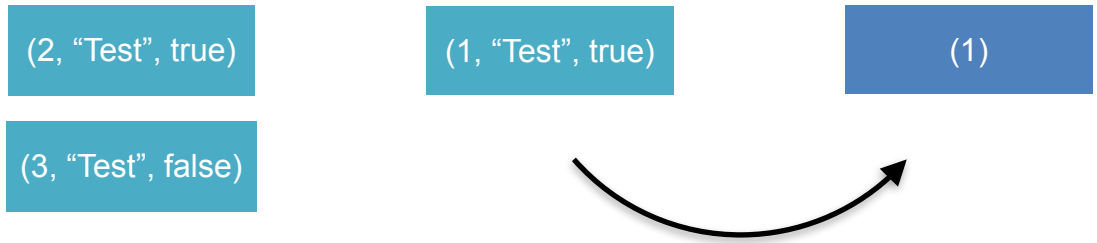
Forward specified attributes

Close

Close data source

Parameters: requested attributes

Next



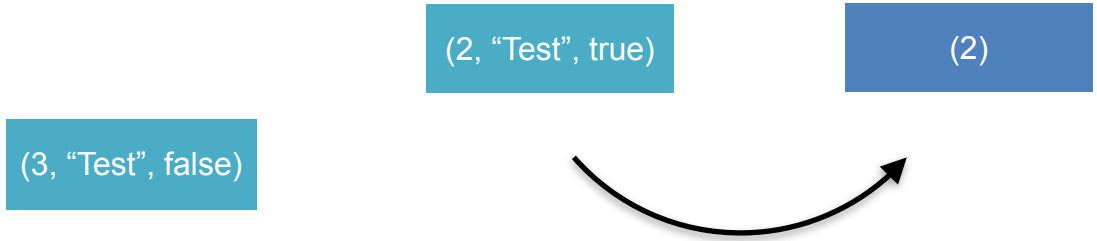
Parameters: 1st attribute

Example: Projection Operator

Open	Next	Close
Open data source	Forward specified attributes	Close data source

Parameters: requested attributes

Next



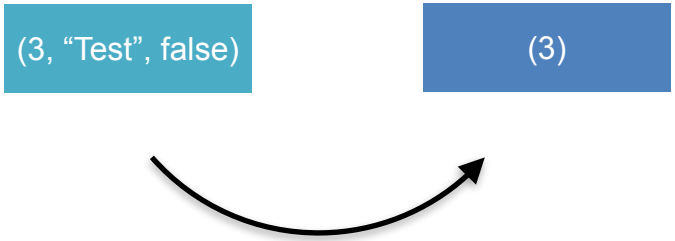
Parameters: 1st attribute

Example: Projection Operator

Open	Next	Close
Open data source	Forward specified attributes	Close data source

Parameters: requested attributes

Next



Parameters: 1st attribute

Example: Projection Operator

Open

Open data source

Next

Forward specified attributes

Close

Close data source

Parameters: requested attributes

Close

Parameters: 1st attribute

Example: Selection Operator

Open	Next	Close
Open data source	Forward next tuple that satisfies comparison	Close data source

Parameters: attribute, comparator, value

Open t1

(1, “Test”, true)

(2, “Test”, true)

(3, “Test”, false)

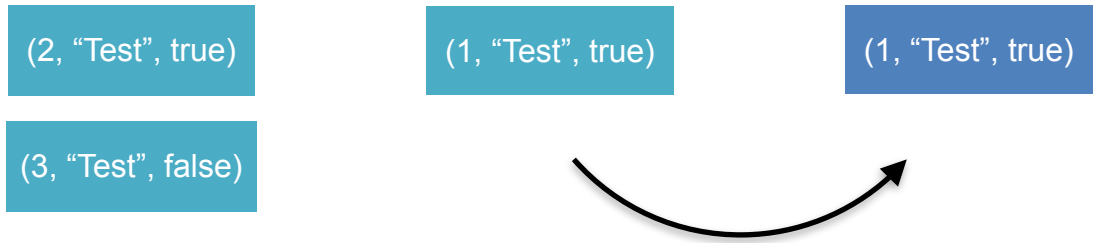
Parameters: 1st attribute, <, 3

Example: Selection Operator

Open	Next	Close
Open data source	Forward next tuple that satisfies comparison	Close data source

Parameters: attribute, comparator, value

Next



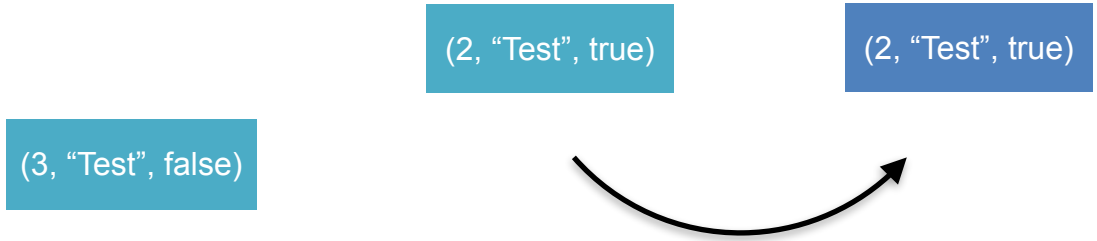
Parameters: 1st attribute, <, 3

Example: Selection Operator

Open	Next	Close
Open data source	Forward next tuple that satisfies comparison	Close data source

Parameters: attribute, comparator, value

Next



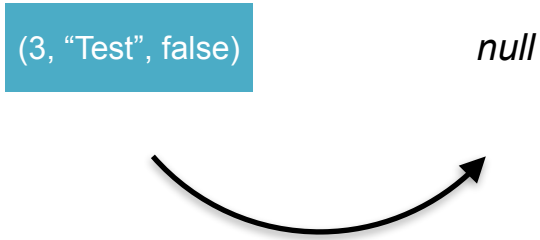
Parameters: 1st attribute, <, 3

Example: Selection Operator

Open	Next	Close
Open data source	Forward next tuple that satisfies comparison	Close data source

Parameters: attribute, comparator, value

Next



Parameters: 1st attribute, <, 3

Example: Selection Operator

Open	Next	Close
Open data source	Forward next tuple that satisfies comparison	Close data source

Parameters: attribute, comparator, value

Close

Parameters: 1st attribute, <, 3

Example: Distinct Operator

Open	Next	Close
Open data source	Forward next new tuple	Close data source

Parameters: -

Open Intermediate Result

(“Test”, true)

(“Test”, true)

(“Test”, false)

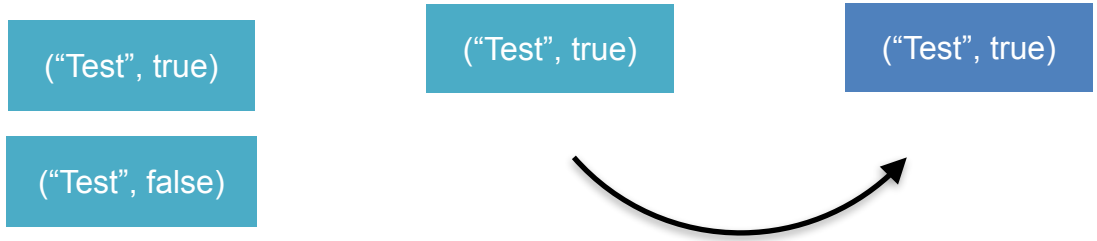
Parameters: -

Example: Distinct Operator

Open	Next	Close
Open data source	Forward next new tuple	Close data source

Parameters: -

Next

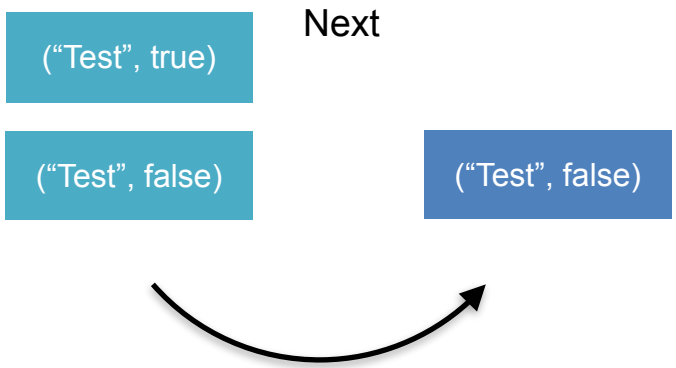


Parameters: -

Example: Distinct Operator

Open	Next	Close
Open data source	Forward next new tuple	Close data source

Parameters: -



Parameters: -

Example: Distinct Operator

Open

Open data source

Next

Forward next new tuple

Close

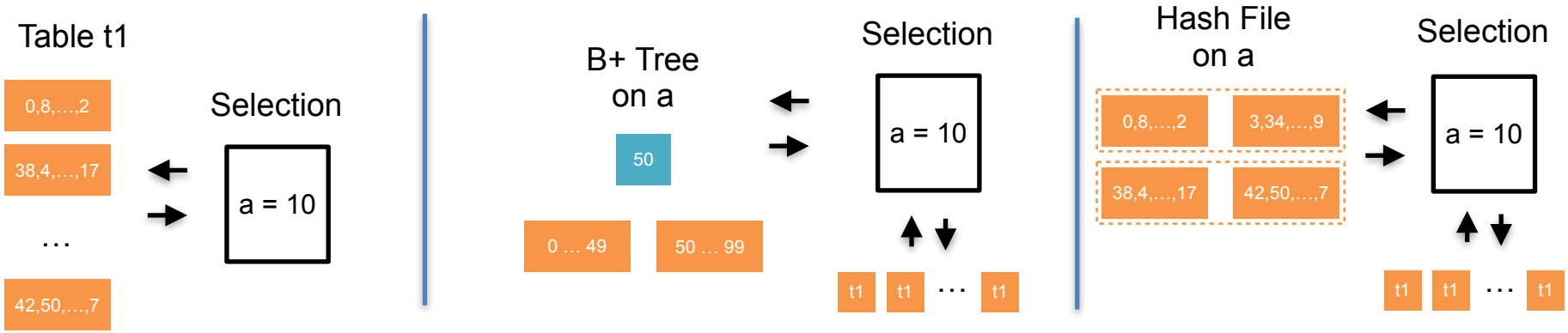
Close data source

Parameters: -

Close

Parameters: -

Task 4: Query Execution Plans



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

Info: Loading any block costs 1 IO

… assume worst case!

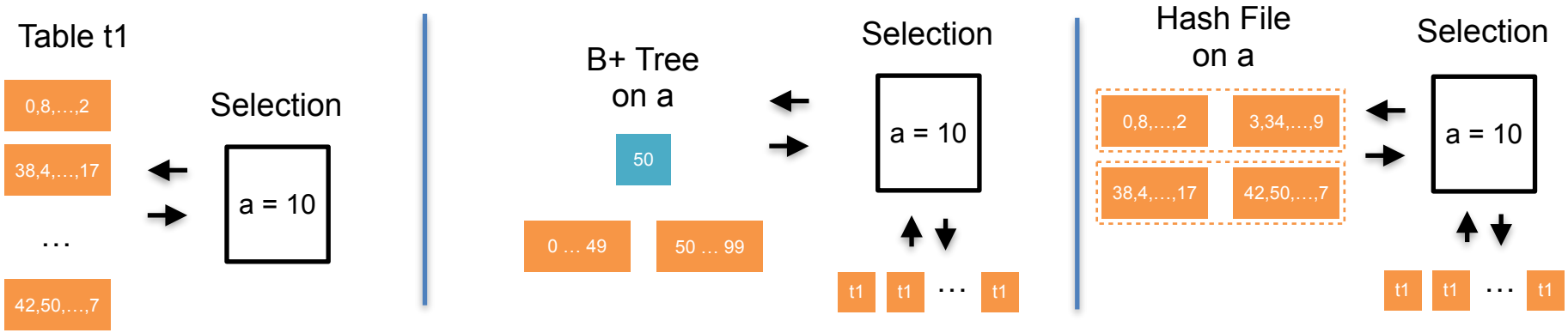
- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a = 10

(A) Table Scanning

(B) B+ Tree + Loading tuples

(C) Hash File + Loading tuples

Task 4: Query Execution Plans



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

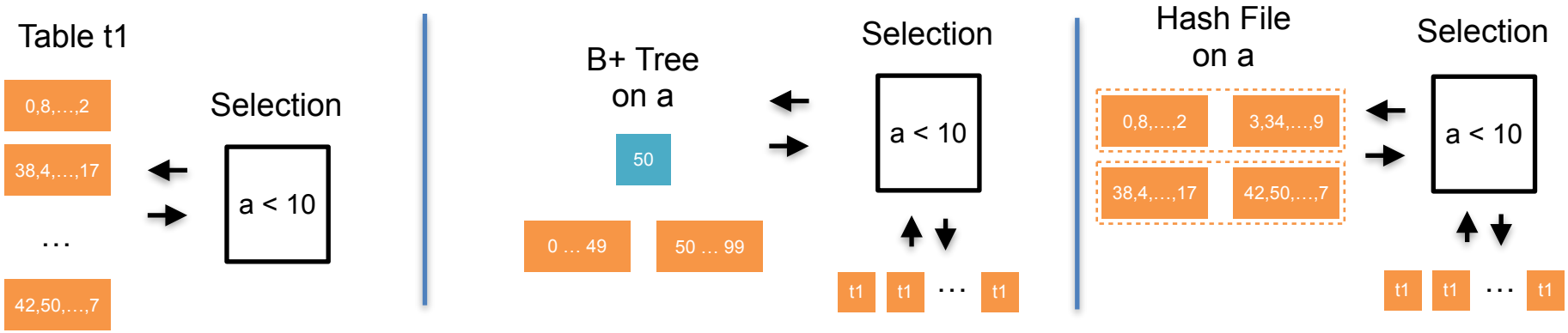
Info: Loading any block costs 1 IO

… assume worst case!

- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a = 10

(A) Table Scanning	(B) B+ Tree + Loading tuples	(C) Hash File + Loading tuples
20 IO	2 + 1 = 3 IO	2 + 1 = 3 IO

Task 5: Query Execution Plans



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

Info: Loading any block costs 1 IO

… assume worst case!

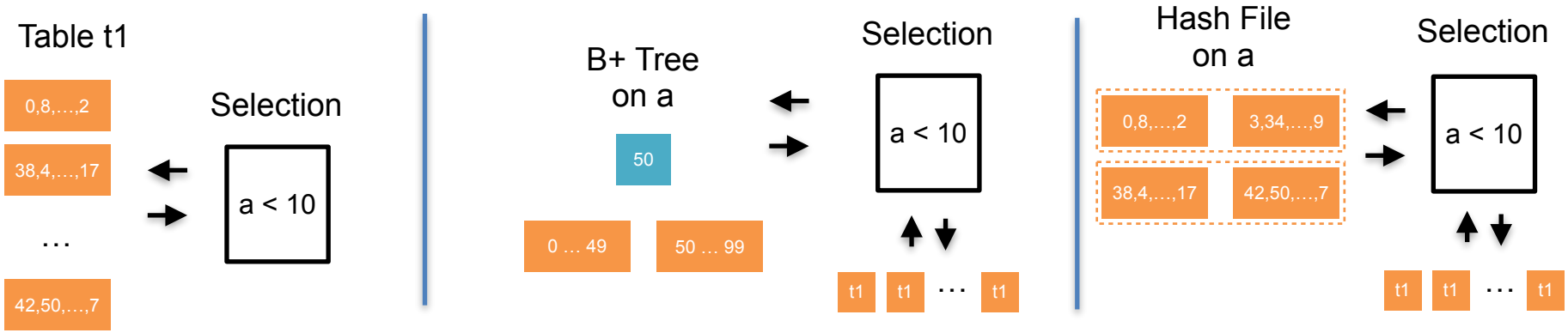
- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a < 10

(A) Table Scanning

(B) B+ Tree + Loading tuples

(C) Hash File + Loading tuples

Task 5: Query Execution Plan



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

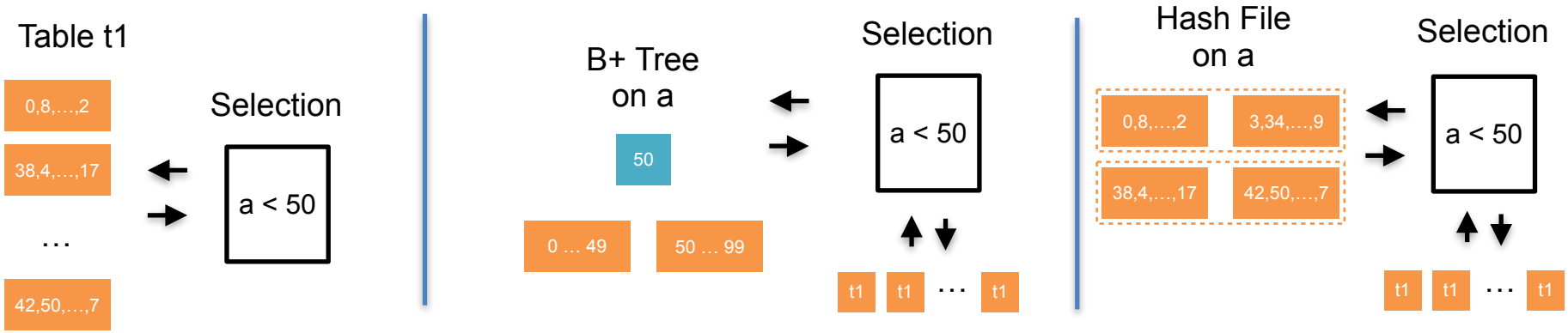
Info: Loading any block costs 1 IO

… assume worst case!

- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a < 10

(A) Table Scanning	(B) B+ Tree + Loading tuples	(C) Hash File + Loading tuples
20 IO	2 + 10 = 12 IO	20 + 10 = 30 IO

Task 6: Query Execution Plans



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

Info: Loading any block costs 1 IO

… assume worst case!

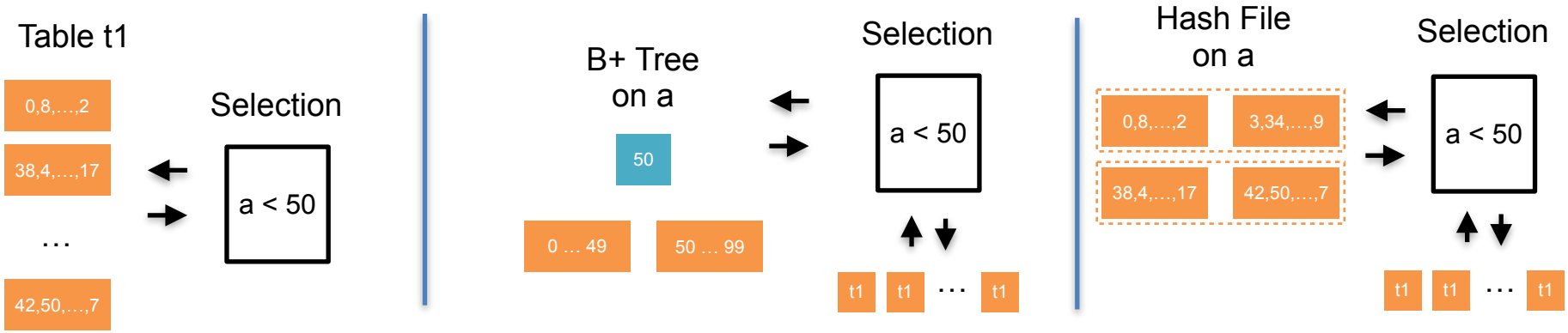
- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a < 50

(A) Table Scanning

(B) B+ Tree + Loading tuples

(C) Hash File + Loading tuples

Task 6: Query Execution Plans



Info: a is primary key of t1 (with 4 attributes) in range of 0 to 99; t1 has 20 data blocks

Info: Loading a random tuple from t1 costs 1 IO

Info: Loading any block costs 1 IO

… assume worst case!

- **Question**: Which of the above query execution strategies needs fewest IOs for the following query: SELECT a,b,c,d FROM t1 WHERE a < 50

(A) Table Scanning	(B) B+ Tree + Loading tuples	(C) Hash File + Loading tuples
20 IO	2 + 50 = 52 IO	100 + 50 = 150 IO