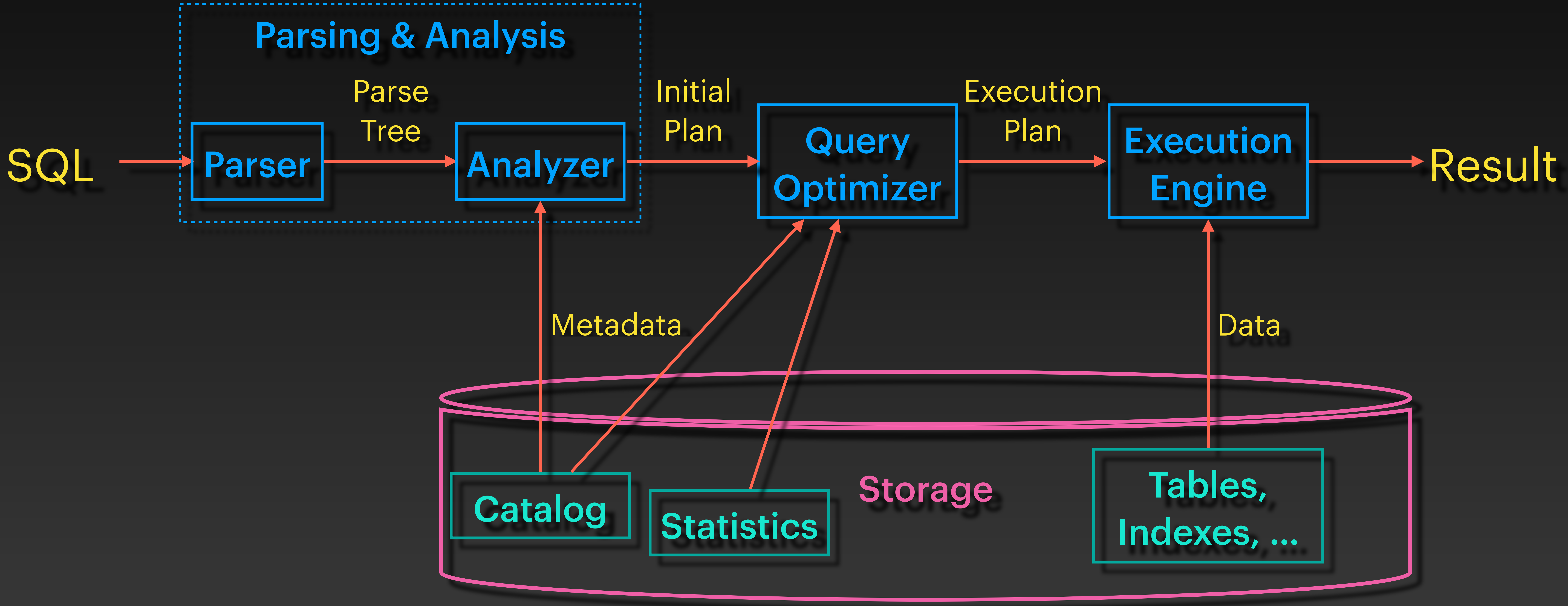
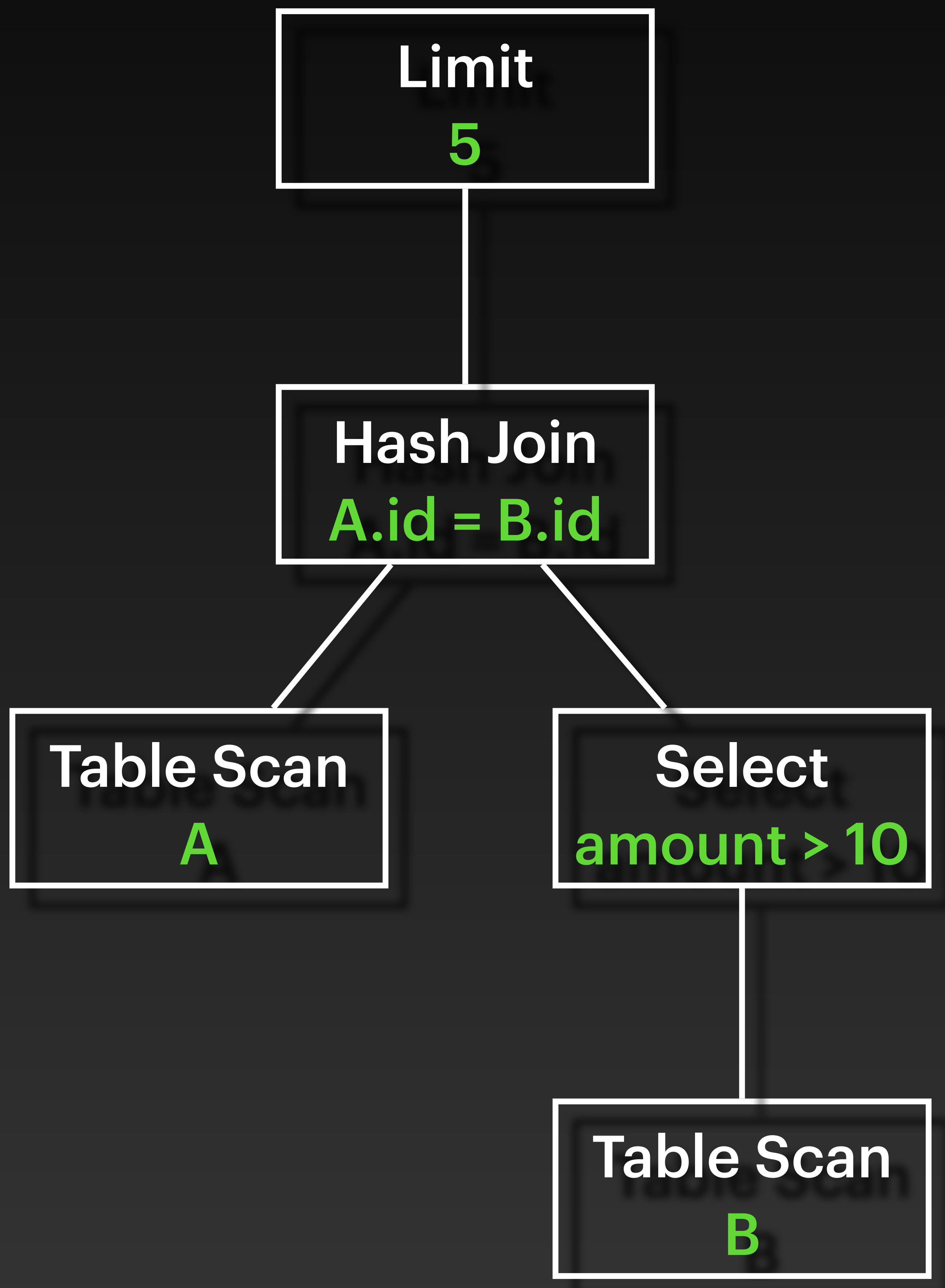

TECH
VAULT
QUERY
EXECUTION
Parsing & Analysis
SQL
Analyzer
Metadata
Execution
Engine
Result
Storage
Statistics
Tables,
Indexes, ...
AMR ELHELW

Query Engine



```sql
SELECT *
FROM A JOIN B
ON A.id = B.id
WHERE B.amount > 100
LIMIT 5
```



```
class TableScan : Node {
   ...
};


class Select : Node {
   ...
};


class HashJoin : Node {
   ...
};


class Limit : Node {
   ...
};


...
```

Materialization Model

- Each operation processes all its input(s) at once, and produces its entire output
- Each Node has a function getResults() which returns all the output of this operation
- A node can call the functions of its input (child) nodes

```
class TableScan : Node {
   Table table;

   ResultSet getResults() {
      ResultSet out;
      for row in table;
         out.add(row);
      return out;
   }
};
```

```
class Select : Node {
   Node child;
   Condition cond;

   ResultSet getResults() {
      ResultSet out;
      for row in child.getResults():
         if check(cond, row):
            out.add(row);
      return out;
   }
};
```

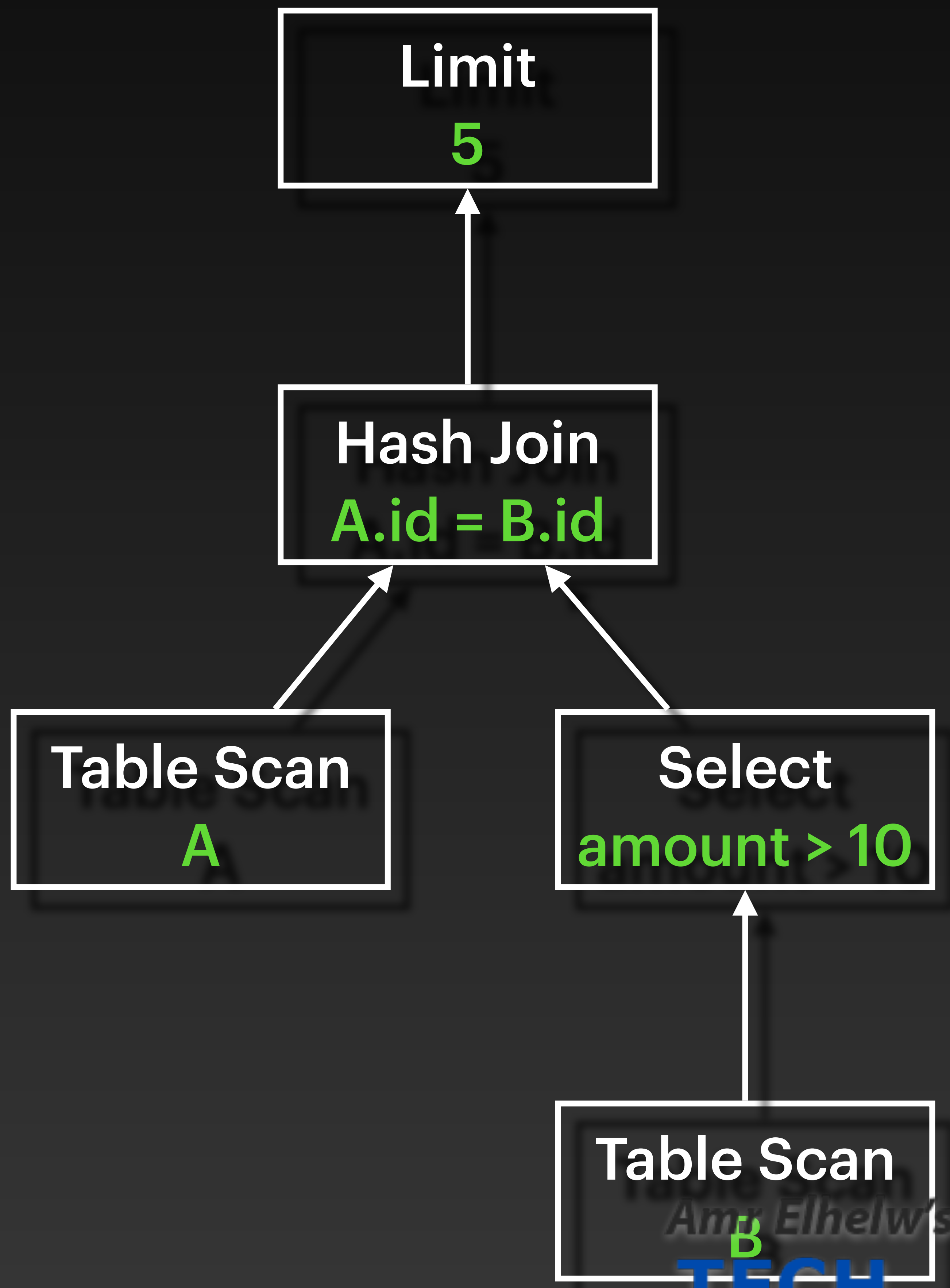
```
class HashJoin : Node {
   Node left, right;
   JoinKey leftKey, rightKey;

   ResultSet getResults() {
      HashTable ht;
      for leftRow in left.getResults():
         ht.add(leftKey, leftRow);

      ResultSet out;
      for rightRow in right.getResults():
        for leftRow in ht.lookup(rightRow):
           out.add(join(leftRow, rightRow));
      return out;
   }
};
```

```
class Limit : Node {
   Node child;
   int n;

   ResultSet getResults() {
      ResultSet out;
      out.addFirst(child.getResults(), n);
      return out;
   }
};
```



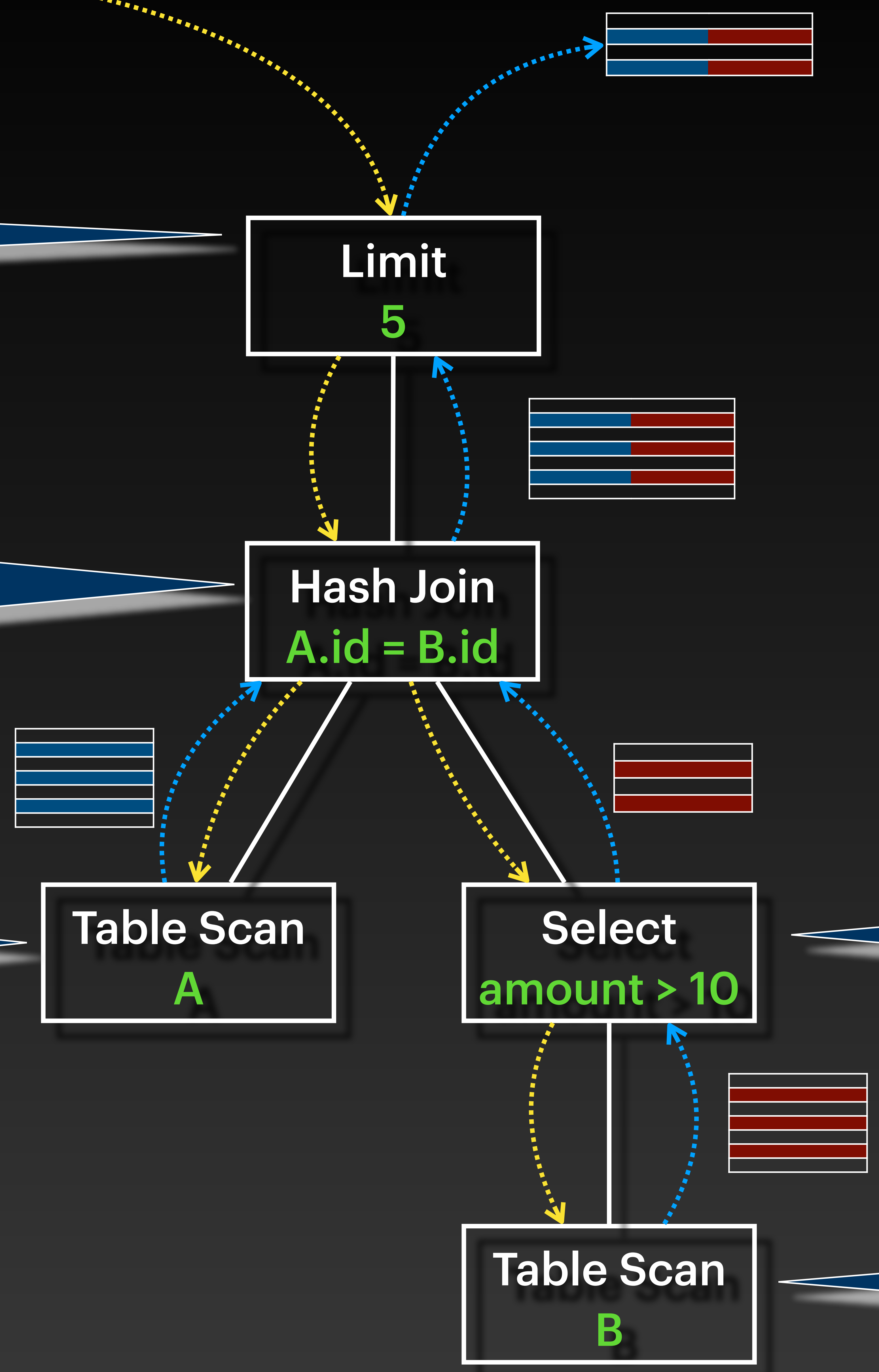
```
root.getResults();
```

```
Limit::getResults() {
   ResultSet out;
   out.addFirst(child.getResults(), n);
   return out;
}
```

```
HashJoin::getResults() {
   HashTable ht;
   for leftRow in left.getResults():
      ht.add(leftKey, leftRow);

   ResultSet out;
   for rightRow in right.getResults():
     for leftRow in ht.lookup(rightRow):
        out.add(join(leftRow, rightRow));
   return out;
}
```

```
TableScan::getResults() {
   ResultSet out;
   for row in table;
      out.add(row);
   return out;
}
```




```
Select::getResults() {
   ResultSet out;
   for row in child.getResults():
      if check(cond, row):
         out.add(row);
   return out;
}
```

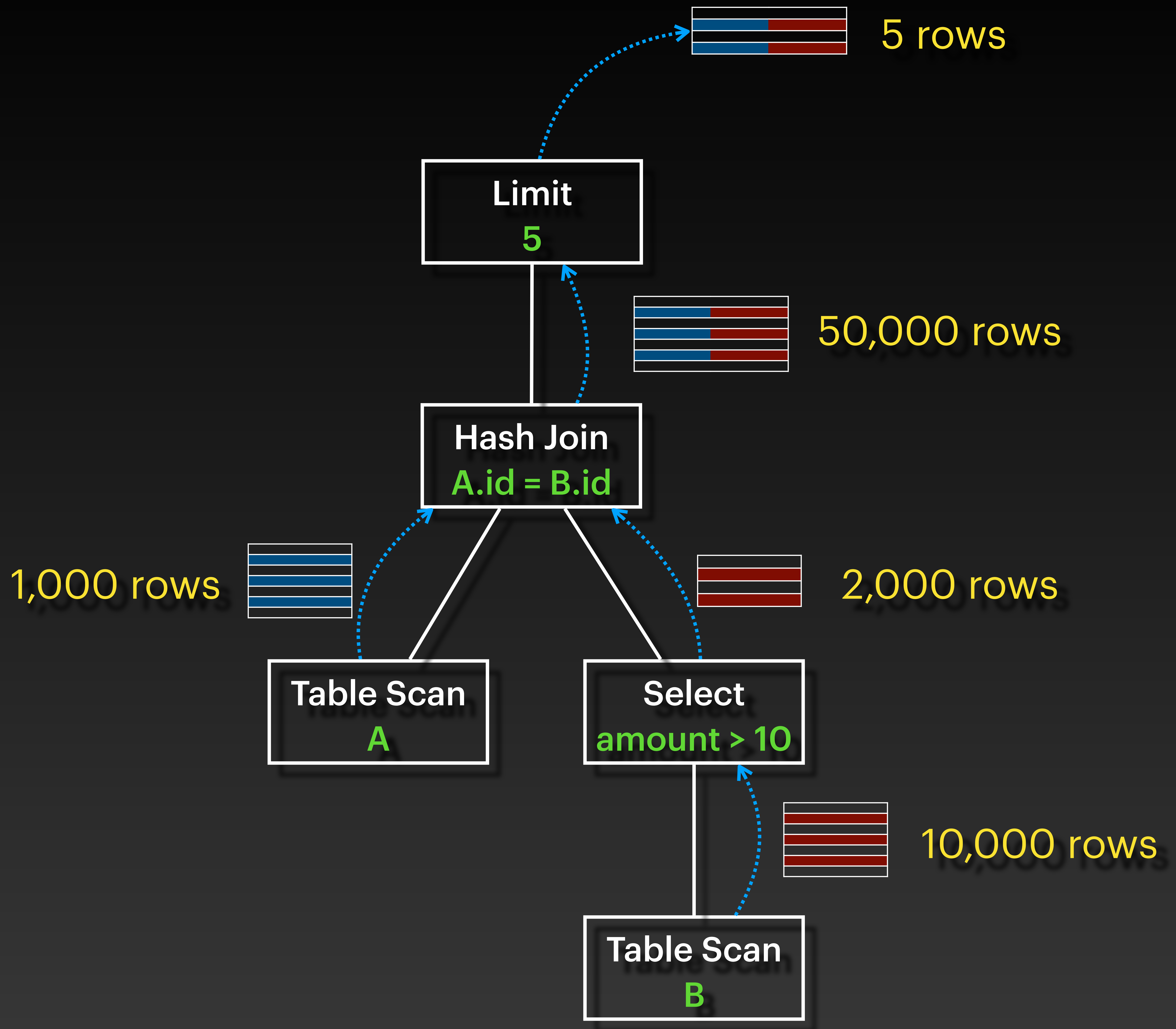
```
TableScan::getResults() {
   ResultSet out;
   for row in table;
      out.add(row);
   return out;
}
```

Advantages

- Simple to implement
- No coordination required between operations
- Each operation is called only once
- Good for OLTP queries
    - Few operations
    - Small intermediate results

Disadvantages

- Not good for OLAP/analytical queries
  - Complex queries, many operations
  - Large intermediate results
- May do some wasted work
  - Example: LIMIT



5 rows
Limit
5
50,000 rows
Hash Join
A.id = B.id
1,000 rows
2,000 rows
Table Scan
A
Select
amount > 10
10,000 rows
Table Scan
B

Iterator Model

- Each operator implements getNext() function
  - Each call returns next row in result, or EOF
- Operators are “active” for longer
  - Need to maintain “state”
- Other important functions:
  - open() - initialization, any work needed before it can start returning results
  - close() - cleanup, release resources, etc.

```
class TableScan : Node {
   open() {
      initializeCursor();
   }

   Row getNext() {
      row = readRowAtCursor();
      if (row == EndOfTable):
         return EOF;
      advanceCursor();
      return row;
};
```

```
class Limit : Node {
   Node child;
   int n;

   Row getNext() {
      if n == 0:
         return EOF;
      row = child.getNext();
      n--;
      return row;
   }
};
```

```
class Select : Node {
   Node child;
   Condition cond;

   Row getNext() {
      while (row = child.getNext()) != EOF {
         if check(cond, row):
            return row;
      }
      return EOF;
   }
};
```

```
class HashJoin : Node {
   Node left, right;
   JoinKey leftKey, rightKey;
   HashTable ht;
   Bool hashTableReady = false;

   Row getNext() {
     if (!hashTableReady) {
        while (leftRow = left.getNext()) != EOF:
           ht.add(leftKey, leftRow);
        hashTableReady = true;
     }
     while (rightRow = right.getNext()) != EOF {
        leftRow = ht.lookup(rightRow):
           return join(leftRow, rightRow);
     }
     return EOF;
   }
};
```


root.getNext();
if n == 0:
  return EOF;
row = child.getNext();
n--;
return row;
Limit
5
if (!hashTableReady) {
   while (leftRow = left.getNext()) != EOF:
      ht.add(leftKey, leftRow);
   hashTableReady = true;
}
while (rightRow = right.getNext()) != EOF {
   leftRow = ht.lookup(rightRow):
   return join(leftRow, rightRow);
}
return EOF;
Hash Join
A.id = B.id
Table Scan
A
Select
amount > 10
while (row = child.getNext()) != EOF {
   if check(cond, row):
      return row;
}
return EOF;
row = readRowAtCursor();
if (row == EndOfTable):
   return EOF;
advanceCursor();
return row;
Table Scan
B
row = readRowAtCursor();
if (row == EndOfTable):
   return EOF;
advanceCursor();
return row;

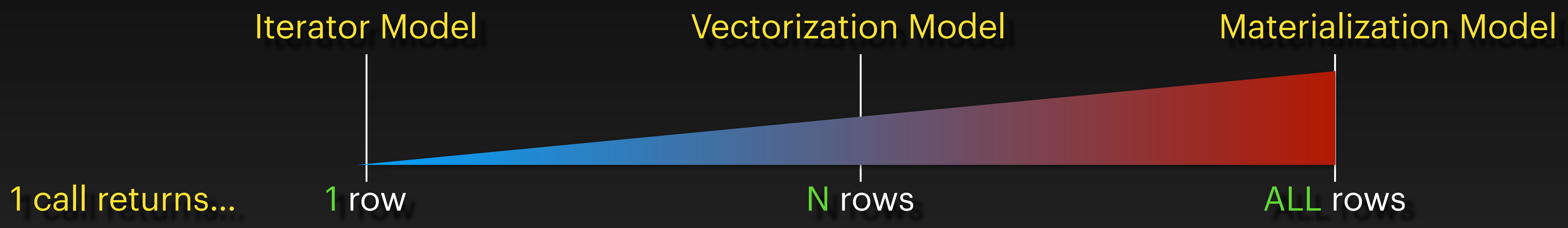
Advantages

- Allows for pipelining
  - Several operators can be “active” at the same time
  - Start producing outputs before reading all the inputs
- Avoid unnecessary data reading (e.g. in the case of LIMIT)
- Does not require keeping all intermediate results in memory
- Most (or all) DBMSs support this model

Disadvantages

- Slightly more complex way to implement
- Lots of context switching between operators, to return one row at a time
- Lots of function calls - can be expensive
- Some operators can be blocking
  - Need to consume all the input before they can start producing output
  - Examples: hash join, aggregation, sort

Vectorization Model



- Each operator implements getNextBatch() function
  - Each call returns next batch of rows in result, or EOF
- Batch size can depend on available memory

```
class TableScan : Node {
   Table table;

   ResultSet getResults() {
      ResultSet out;
      for row in table;
         out.add(row);
      return out;
   }
};
```

Materialization Model

```
class TableScan : Node {
   open() {
      initializeCursor();
   }

   Row getNext() {
      row = readRowAtCursor();
      if (row == EndOfTable):
         return EOF;
      advanceCursor();
      return row;
};
```

Iterator Model

```
class TableScan : Node {
   open() {
      initializeCursor();
   }

   ResultSet getNextBatch() {
      ResultSet out;
      for (i = 0; i < batchSize; ++i) {
         row = readRowAtCursor;
         if (row == EndOfTable):
            break;
         advanceCursor();
         out.add(row);
      }
      if (out.empty()):
         return EOF;
      else
         return out;
};
```

Vectorization Model

Advantages

- Doesn’t need to produce ALL outputs of each operation before executing the next one
  - Will start producing results before reading entire inputs
- Each operation produces a batch that can “usually” fit in memory
- Can work with large datasets (unlike materialization model)
- Can work on complex queries with many operations operators don’t have to wait for too long
- Fewer function calls than iterator model

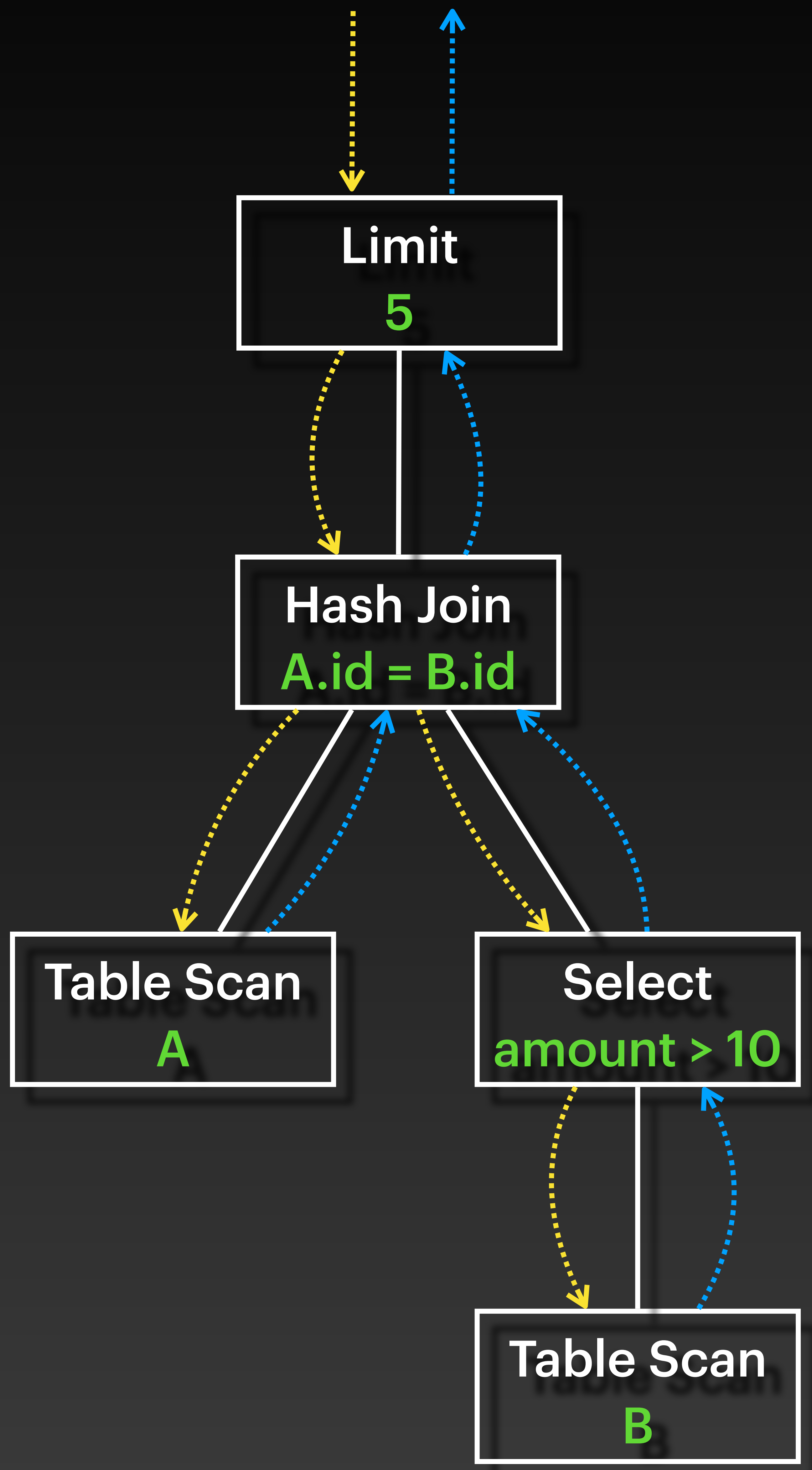
Disadvantages

- More complex implementation
    - We have to think about batches in every operation
- Longer time (than iterator model) until the first output is produced
- “Some” wasted work potentially, but not as much as with materialization

Pull-based vs Push-based

Pull-based Processing

- Top-down
- Each node "pulls" data from its children
- While a node is executing, everything else is waiting (single thread)
- (+) simple to implement & debug
- (-) no parallelism



Push-based Processing

- Bottom-up
- Buffers are added between nodes
- Each node "pushes" data to its parent
- (+) Multiple nodes can work in parallel
- (-) More complex