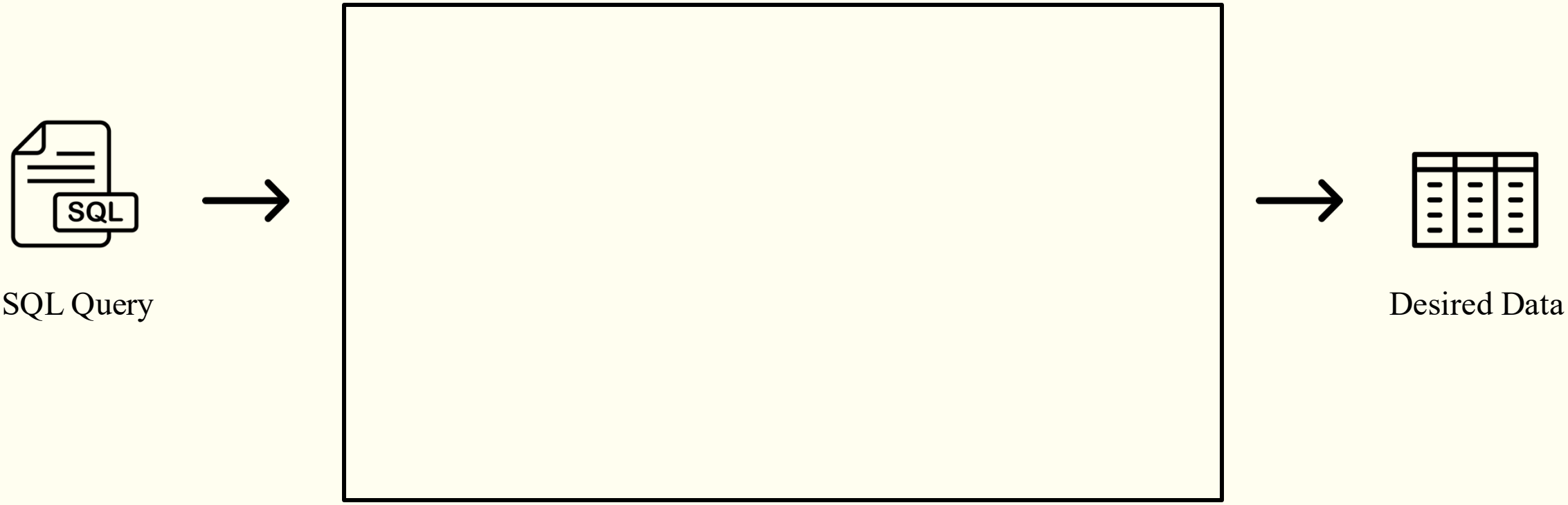
Data Chunk Compaction in Vectorized Execution

Yiming Qiao, Huanchen Zhang

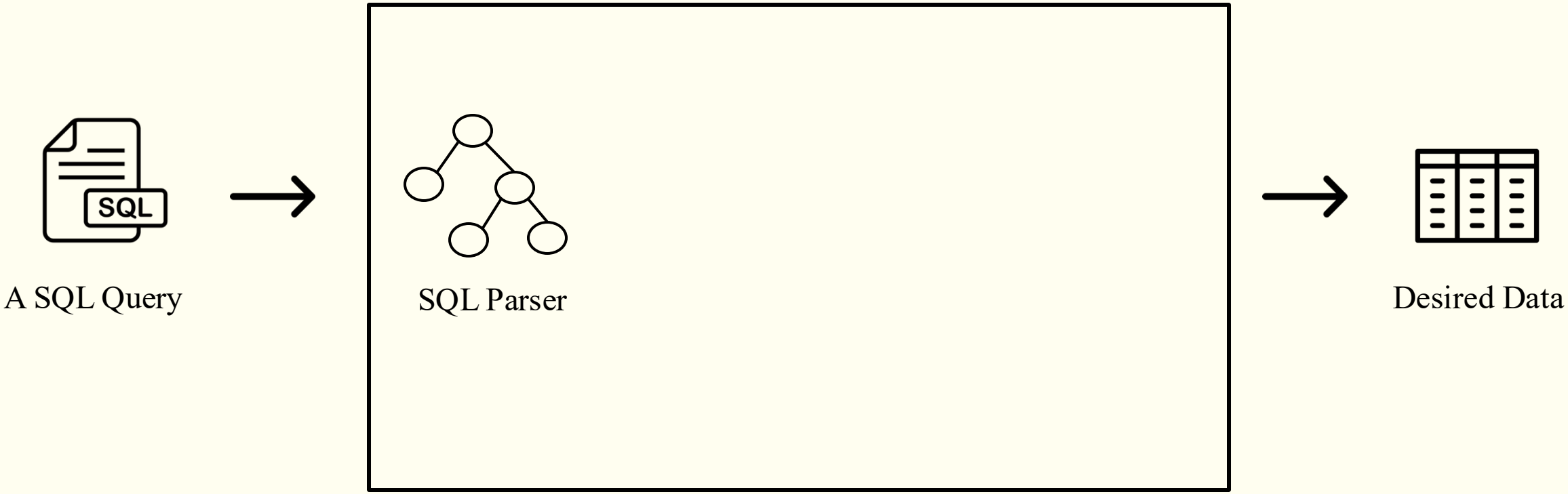
Institute for Interdisciplinary Information Sciences

Tsinghua University

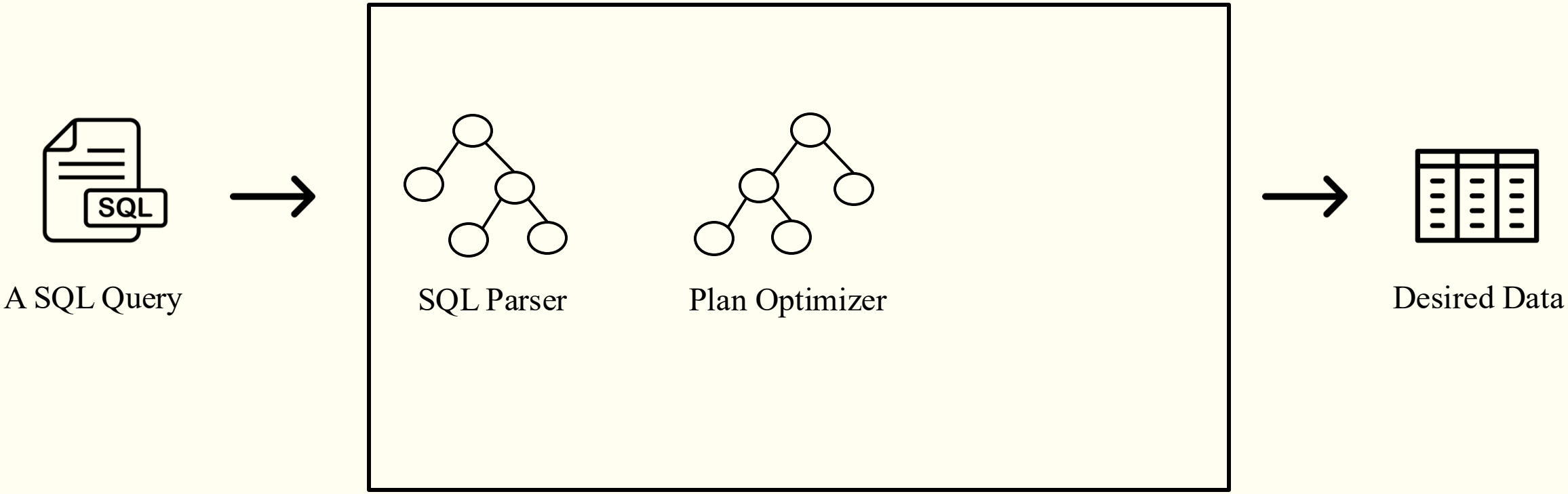
Database System



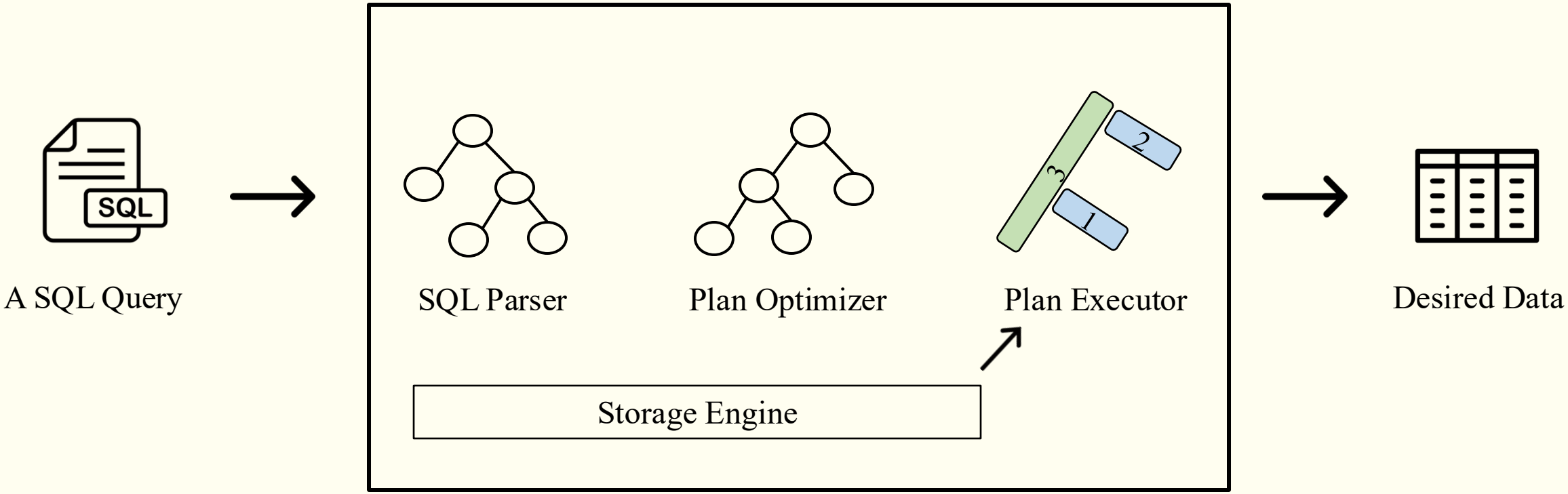
Database System



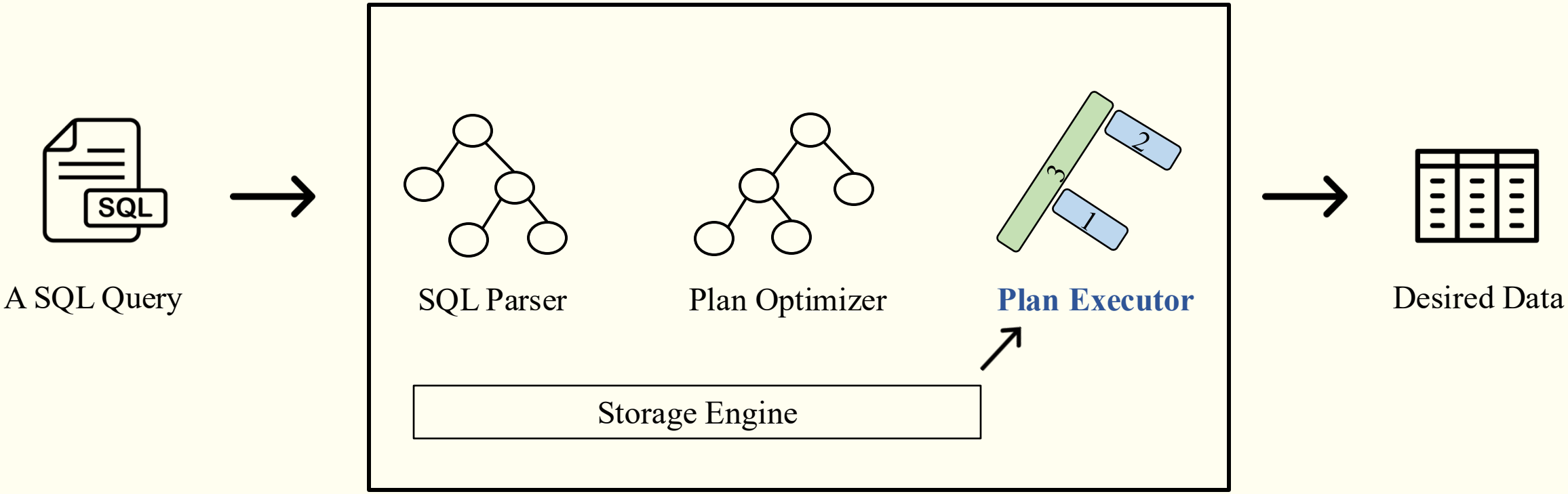
Database System



Database System



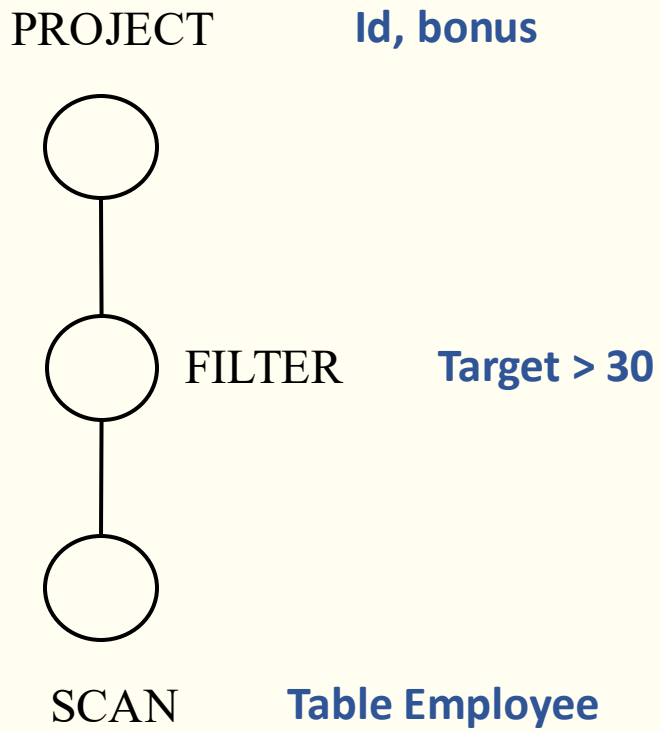
Database System



We Focus on Improving the Query Plan Executor

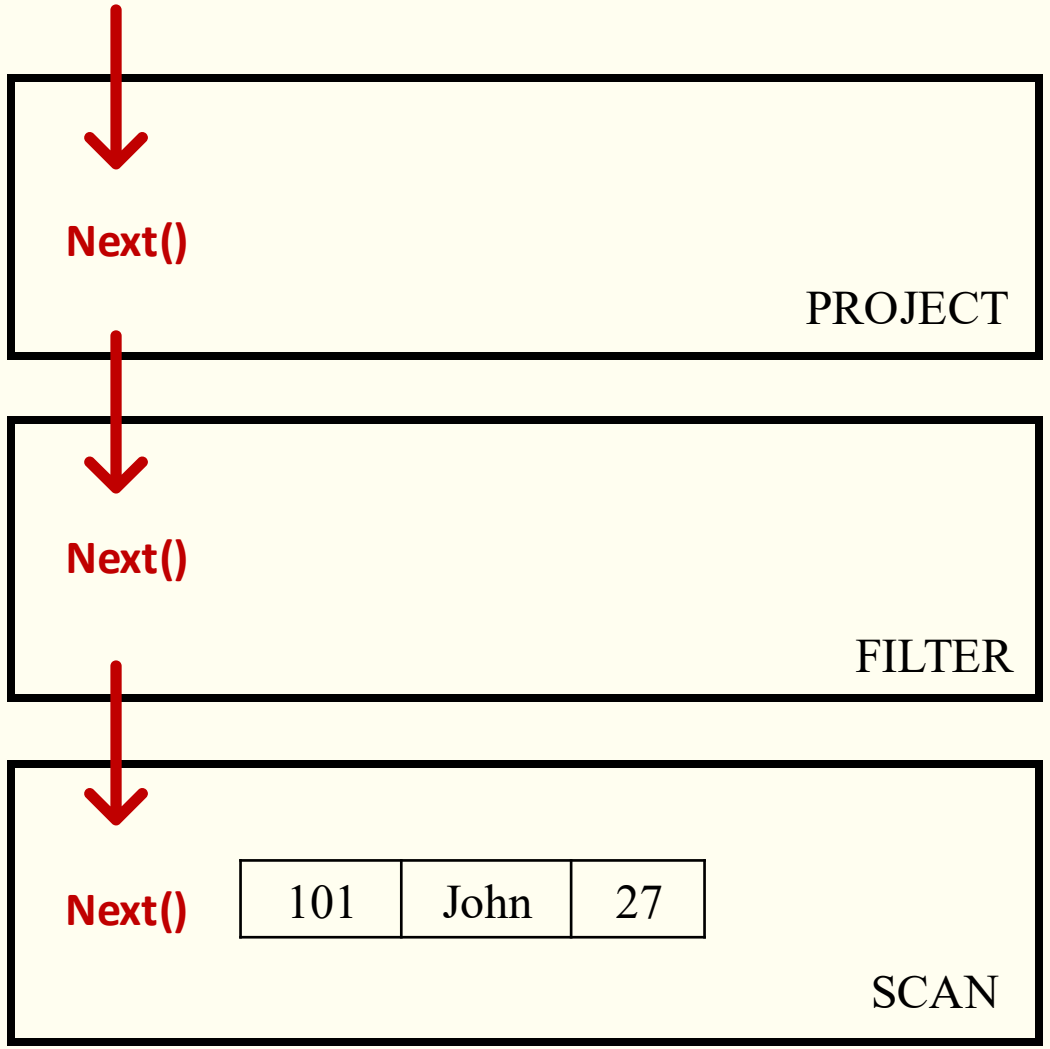
Iterator Execution Model

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



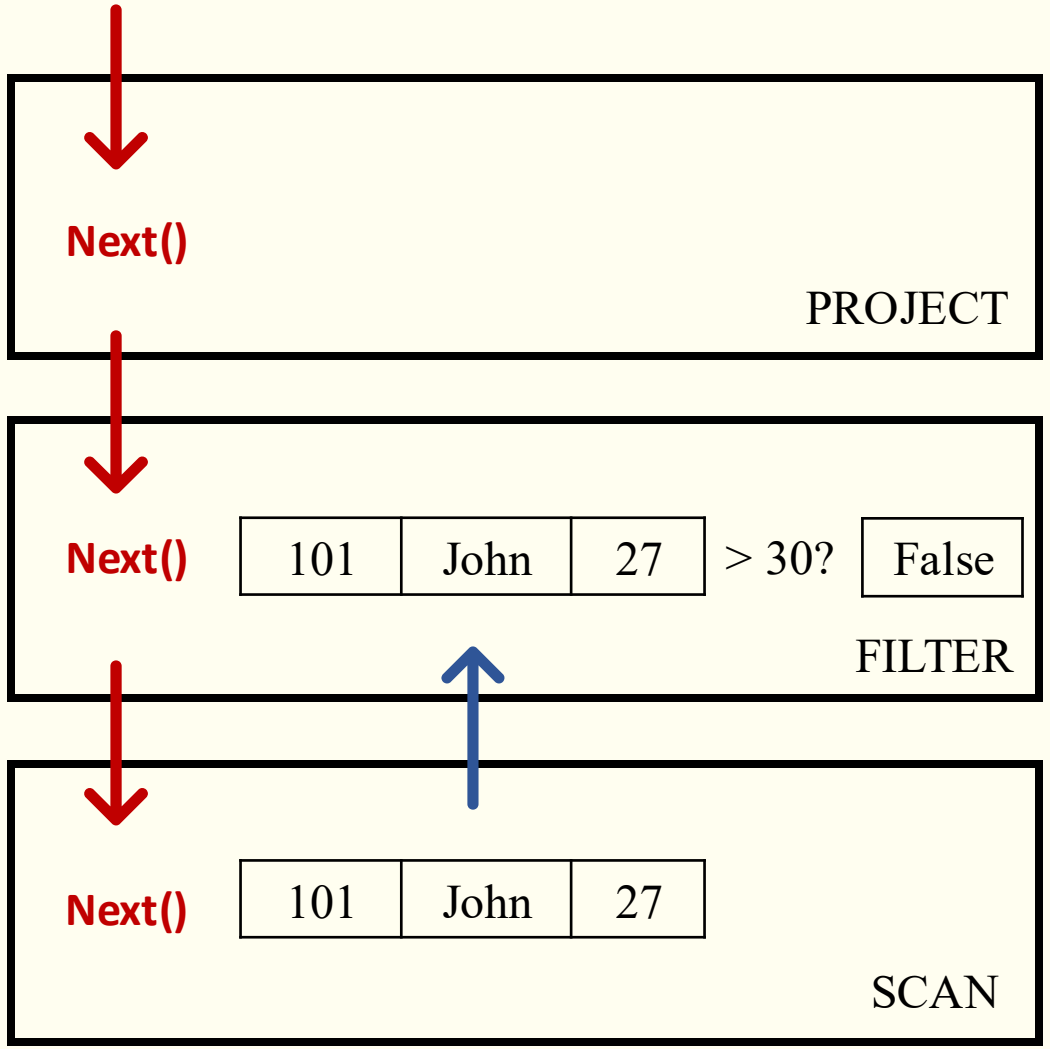
Iterator Execution Model

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



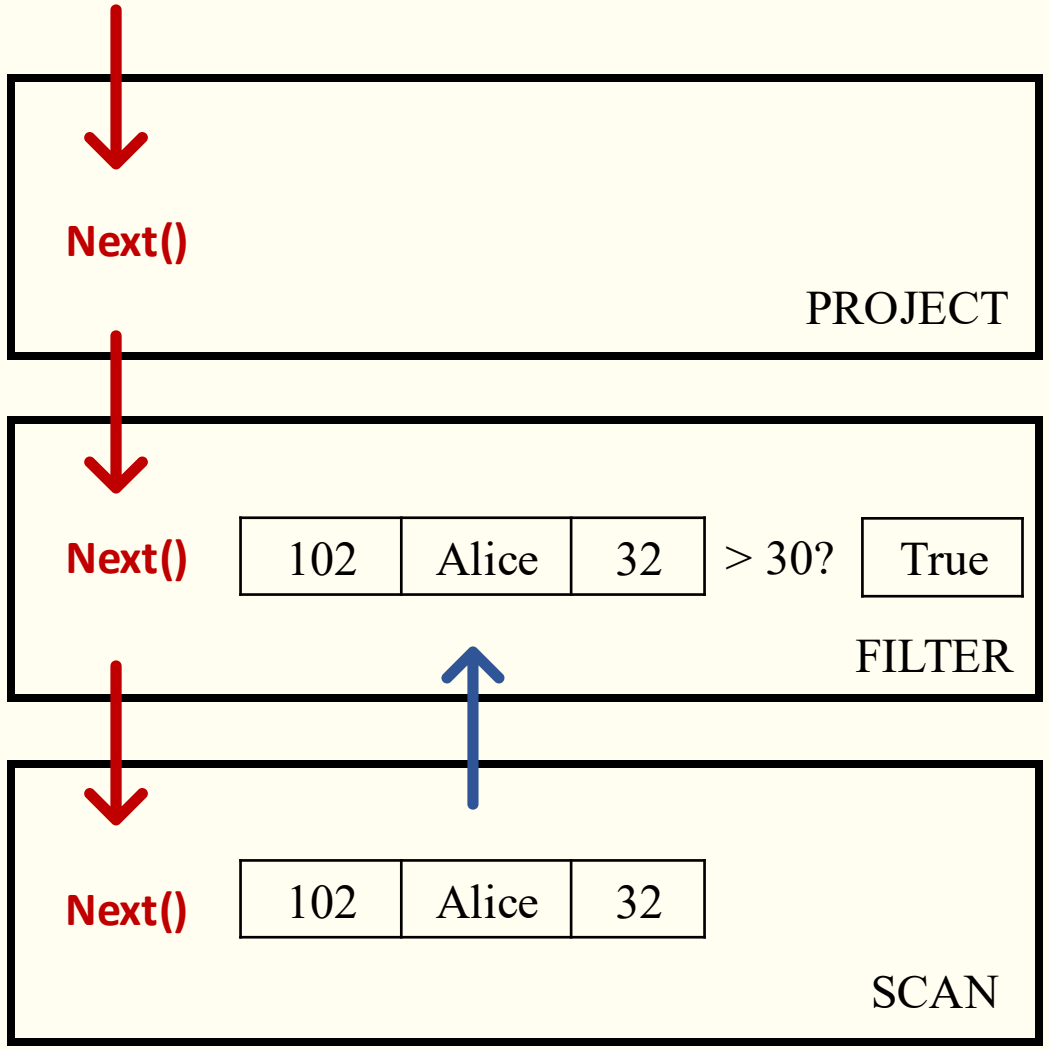
Iterator Execution Model

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



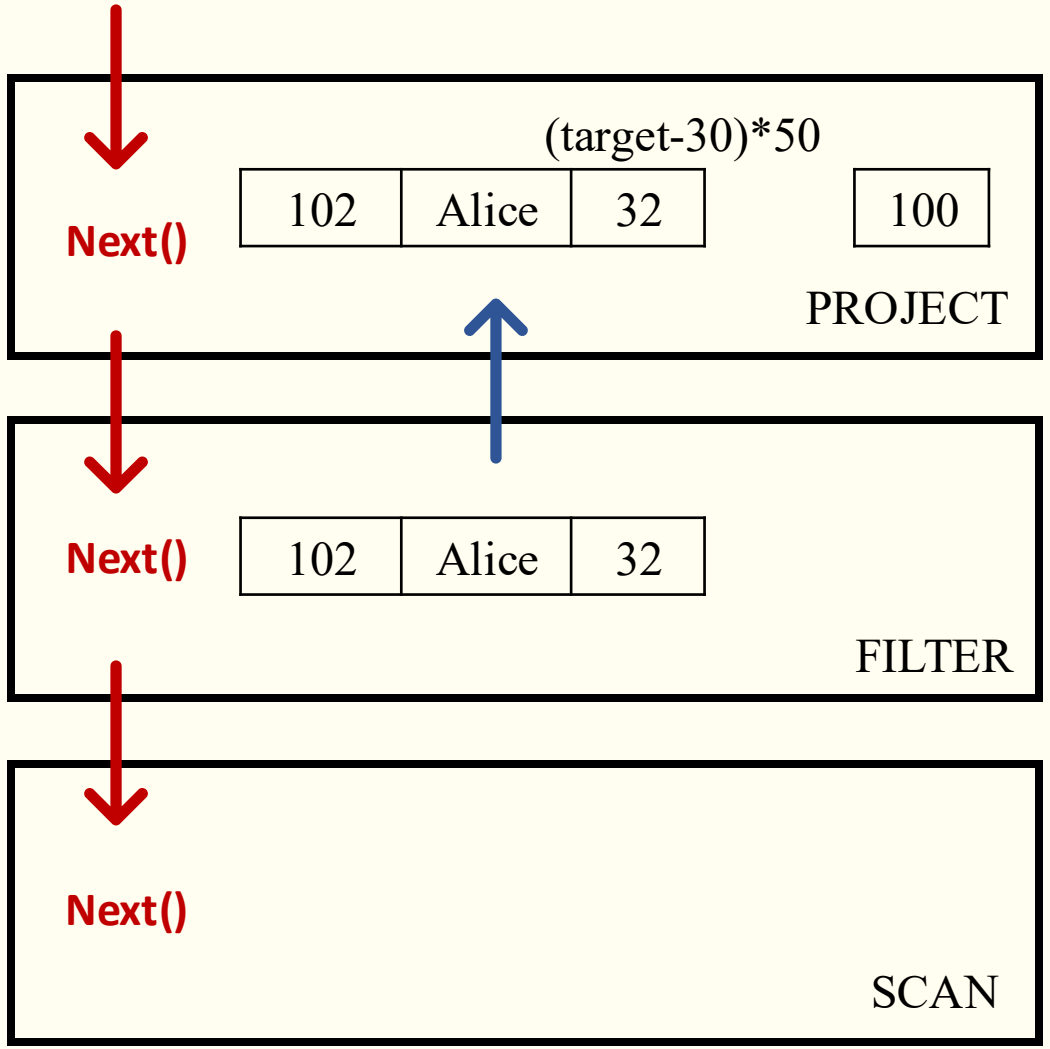
Iterator Execution Model

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



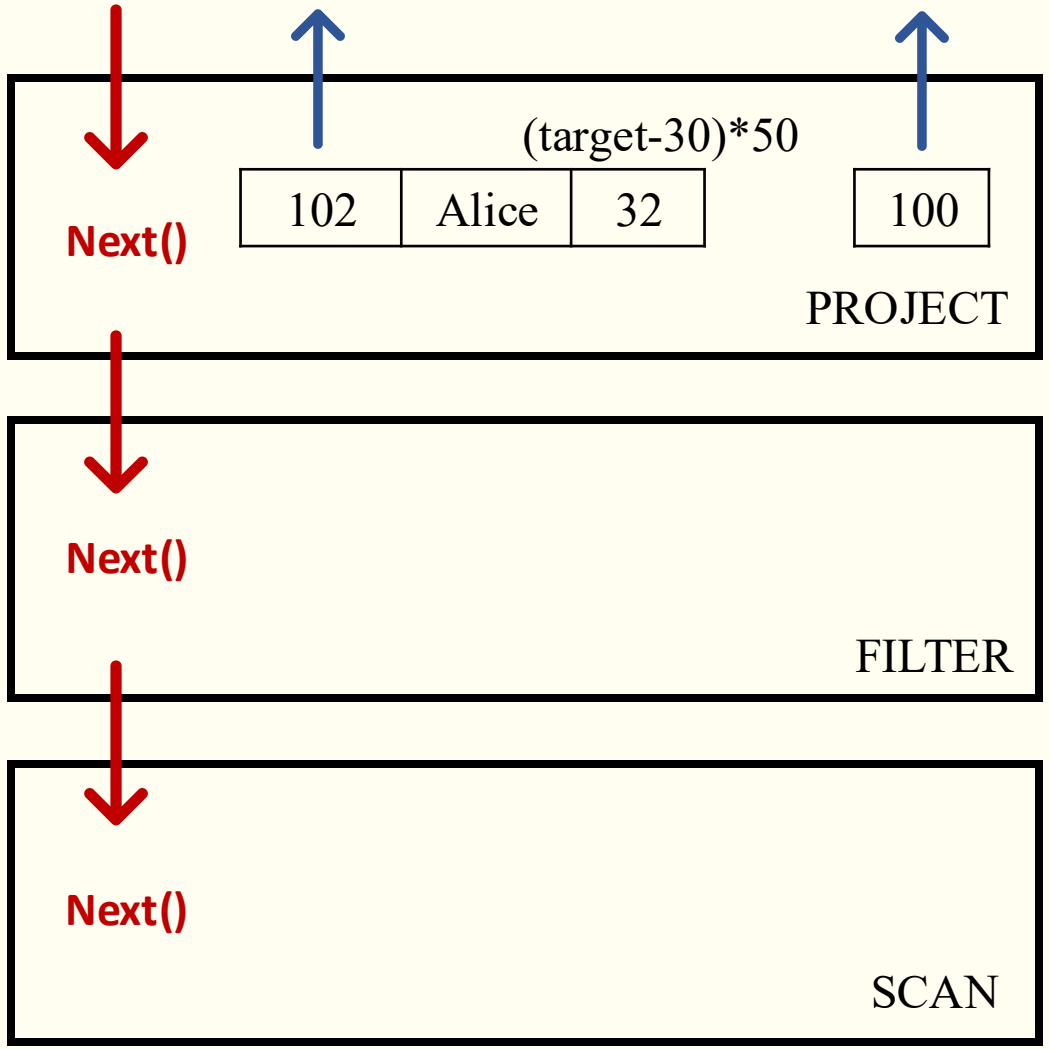
Iterator Execution Model

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



Iterator Execution Model

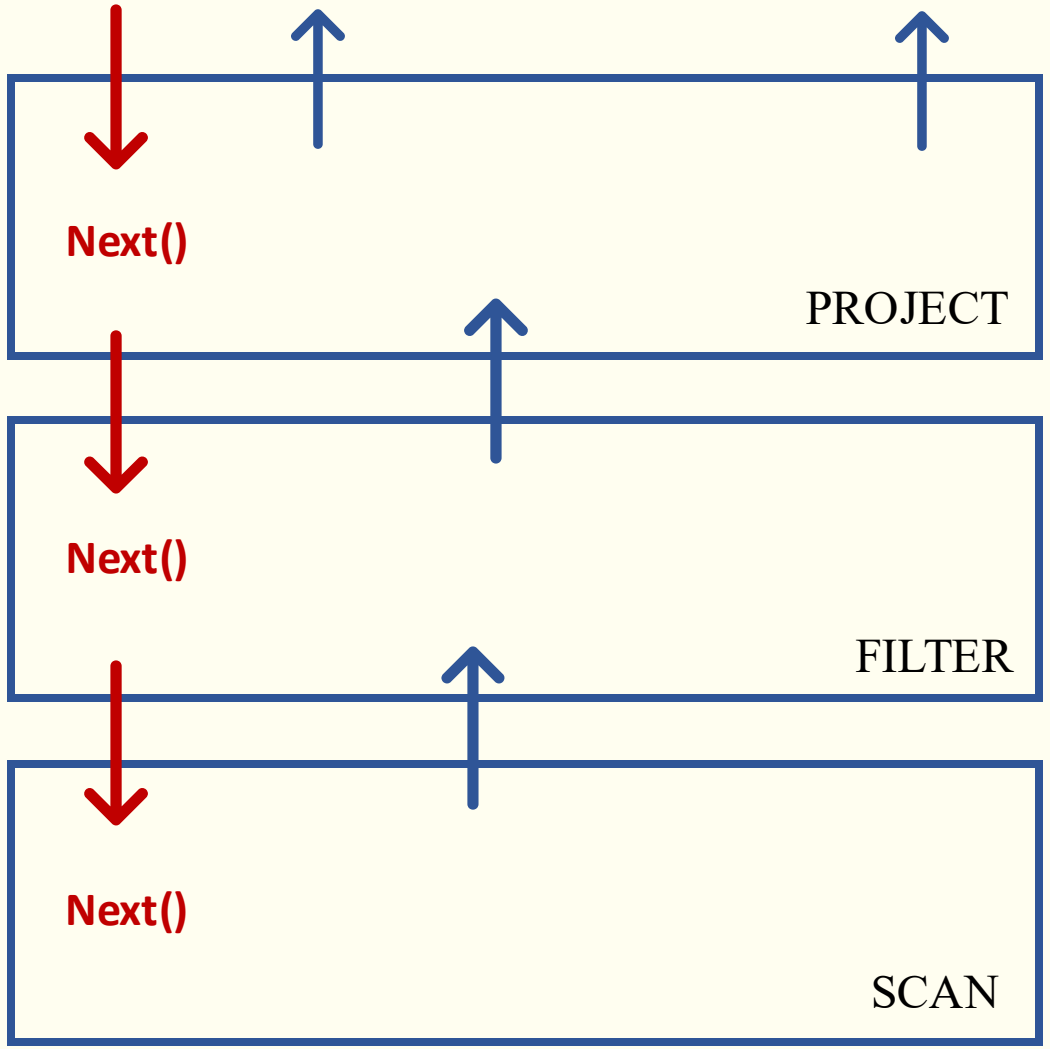
```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```



Iterator Execution Model

- Operators: SCAN, FILTER, PROJECT
- Iterator Interface:
  - **Open()**
  - **Next()** # "Next Tuple Please"
  - **Close()**

Three Function Calls for Processing a Single Tuple!



Buying Beer for a Party. The Silly Way.

An Intuitive Example From Peter Boncz

- Go to a store
- Take one beer bottle
- Pay at the register
- Return home
- Put the bottle in the fridge
- Repeat… 100 times!

= “tuple”

Buying Beer for a Party. A Better Way.

An Intuitive Example From Peter Boncz

- Go to a store
- Take two crates of beer (2x24!)
- Pay at the register
- Return home
- Put crates in the fridge
- Repeat 48x less!

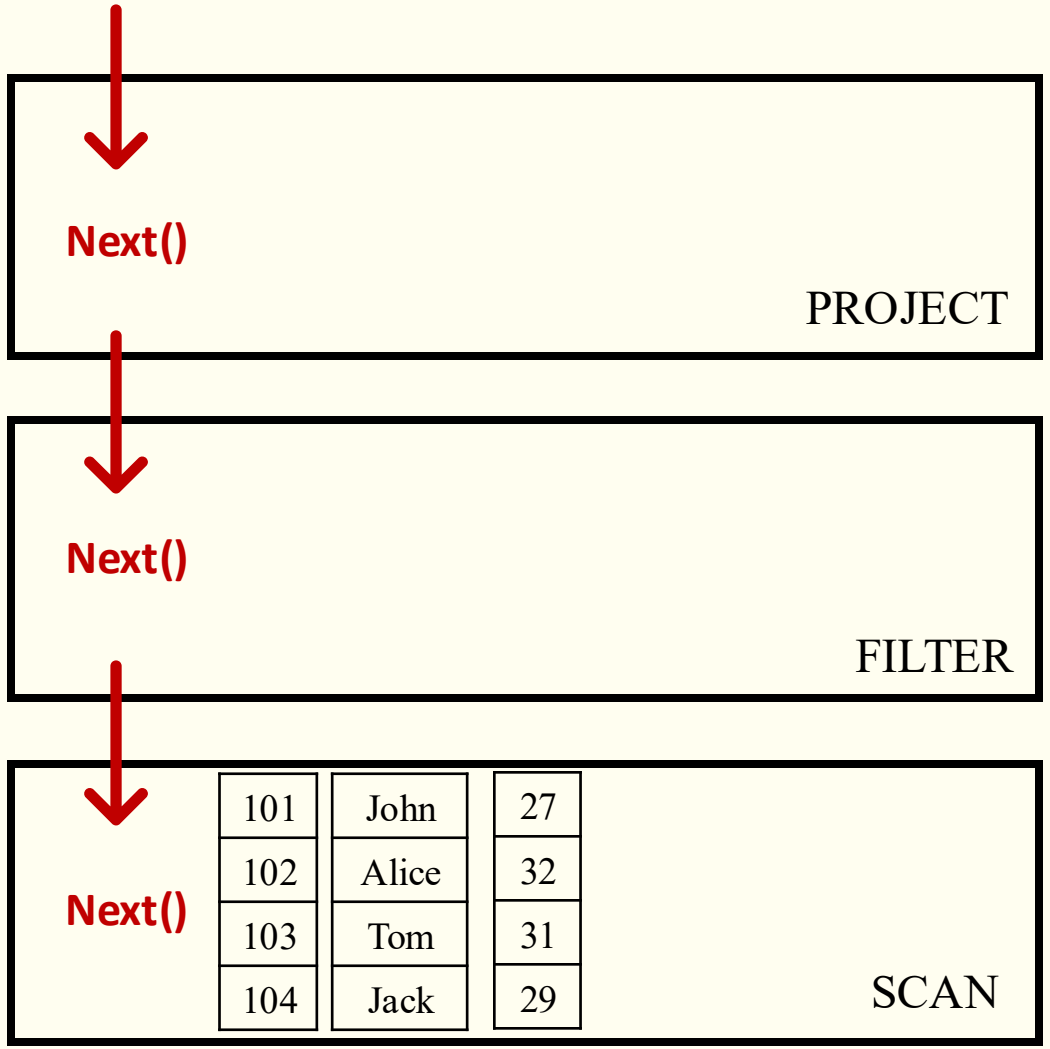
= “vector”

48x less Function Calls for Processing a Single Tuple!

A Smart Way: Vectorized Execution

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```

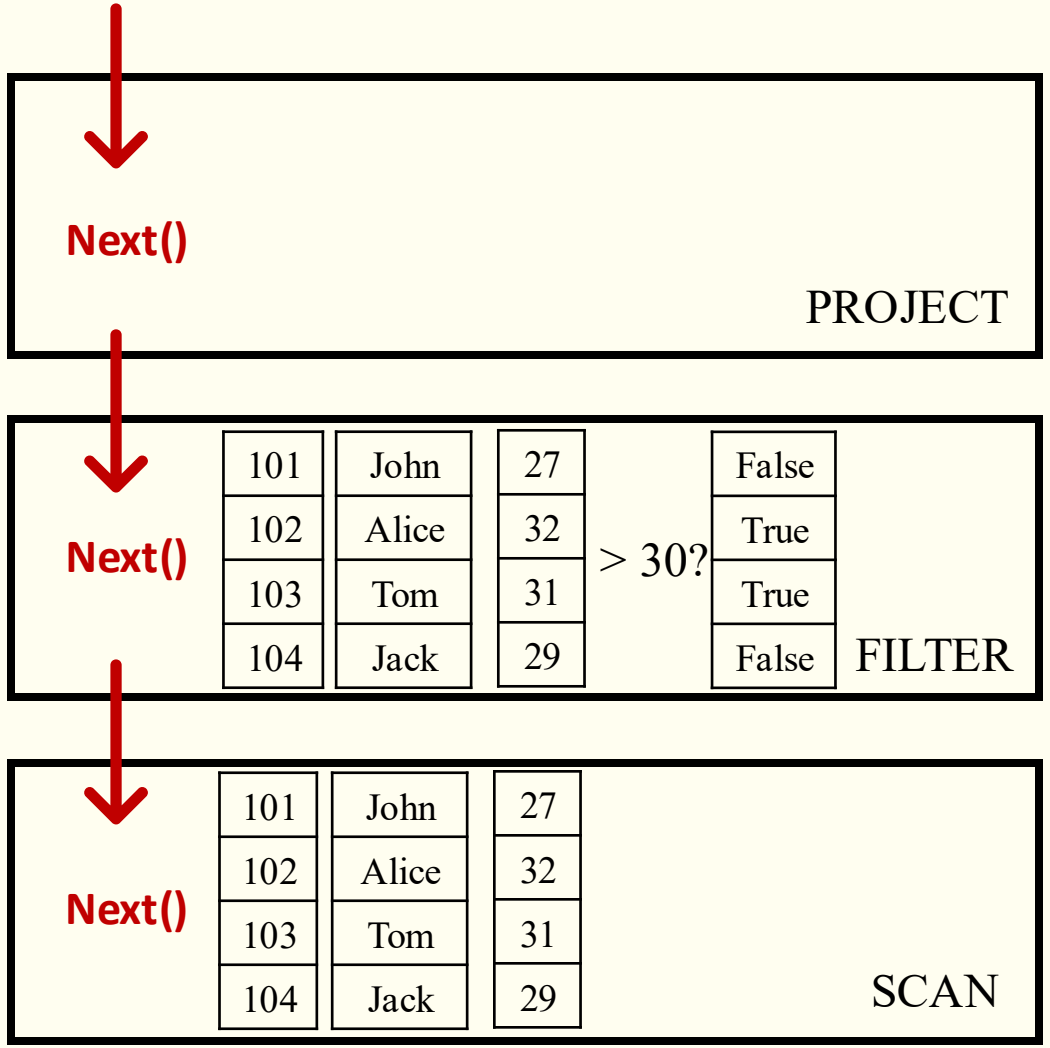
Process a Chunk of Tuples at a Time, with Columnar Store



A Smart Way: Vectorized Execution

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```

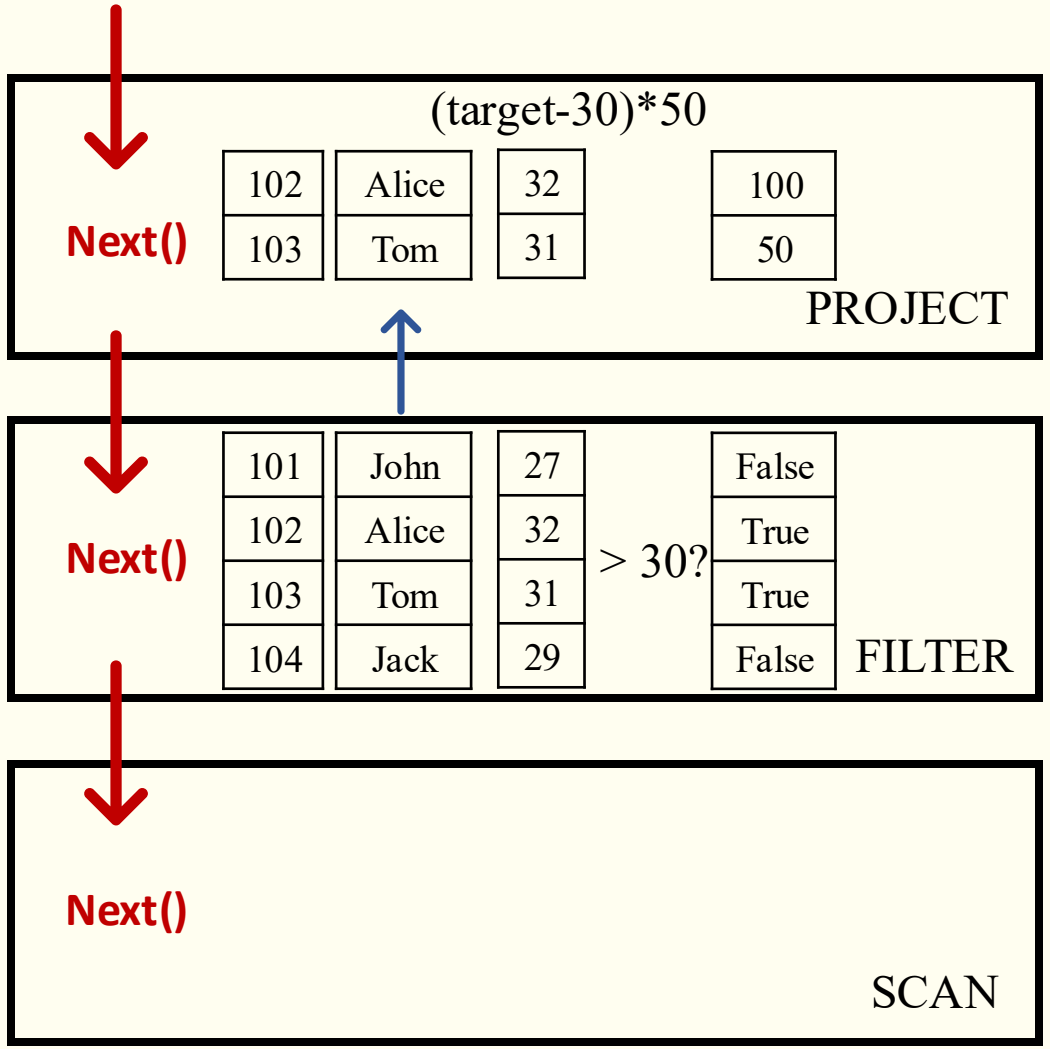
Process a Chunk of Tuples at a Time, with Columnar Store



A Smart Way: Vectorized Execution

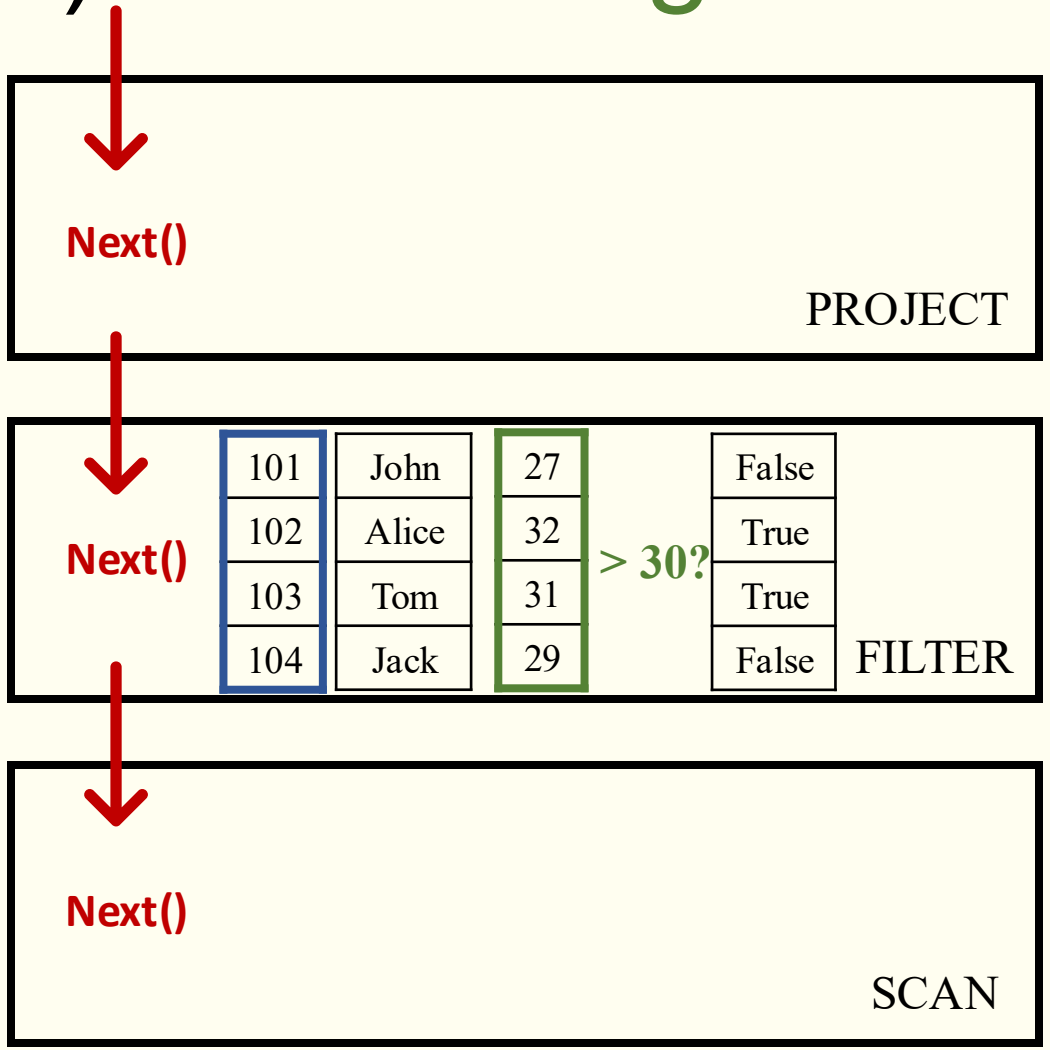
```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```

A Selection Vector/Bit Map is used to Mark Tuples



Vectorized (In-Cache) Processing

- Vector-at-a-time:
  - About 1000 tuples
- Processed in a tight loop:
  - >, +, *
- Cache-friendly:
  - A vector is sequentially accessed

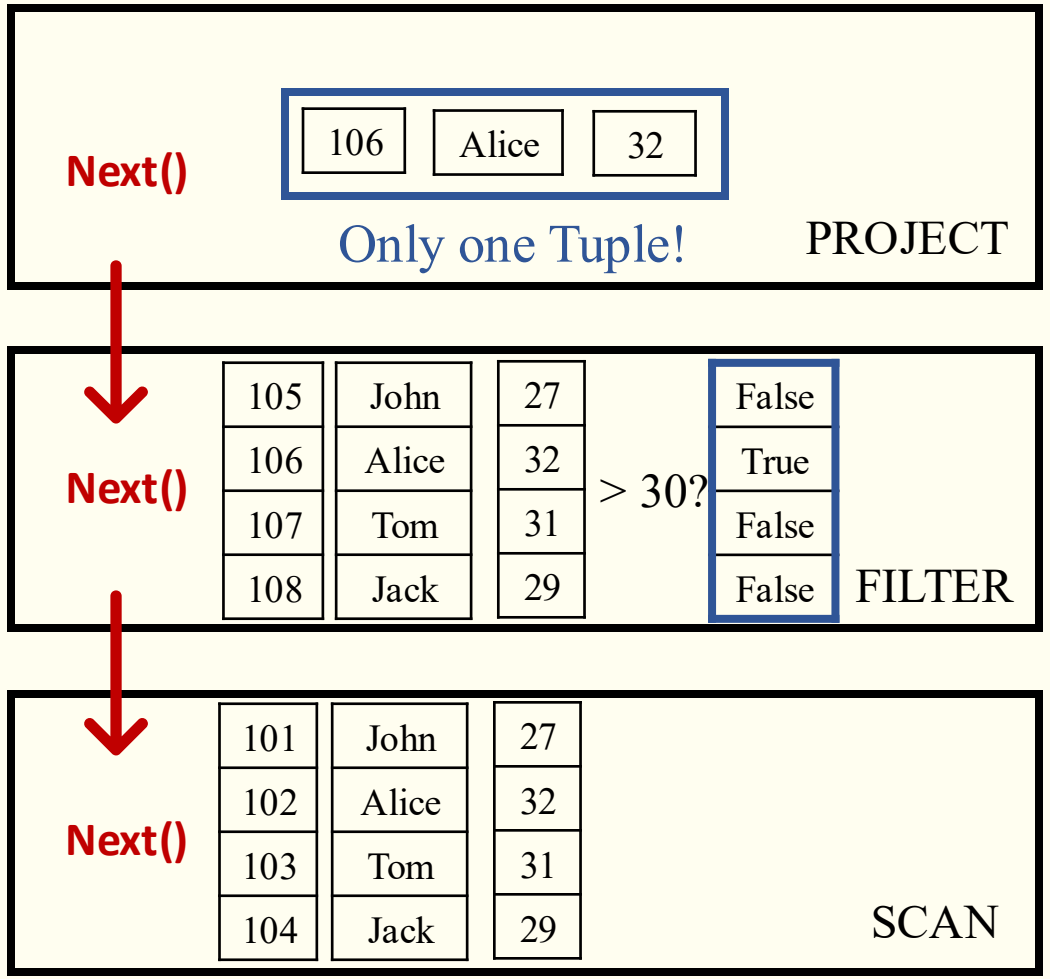


Part 1

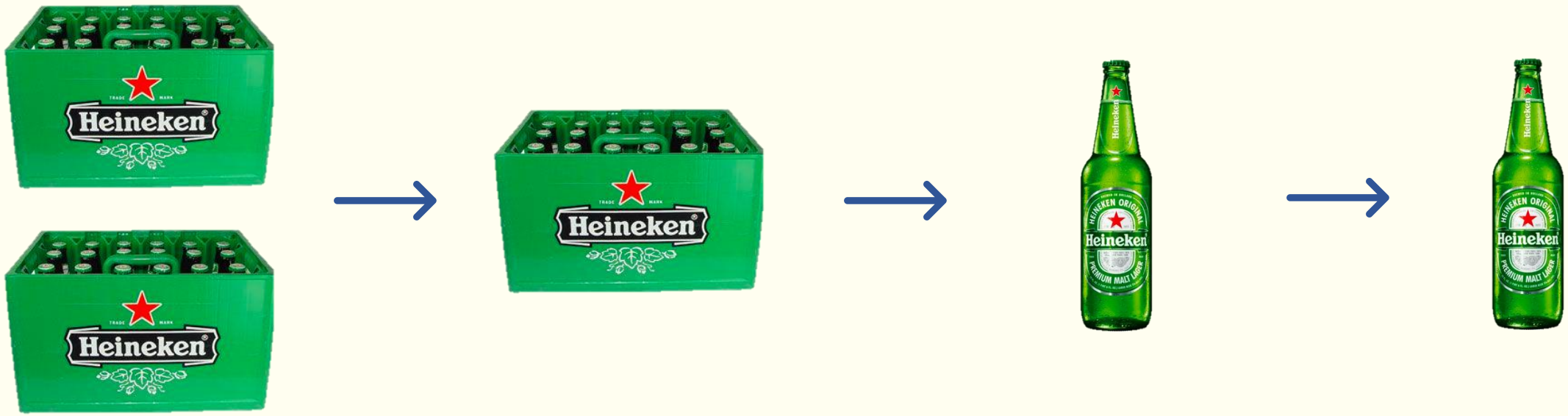
Data Chunk Compaction in Vectorized Execution

Small Chunk Problem

```
SELECT  id,
        (target - 30) * 50 AS bonus
FROM    employee
WHERE   target > 30
```

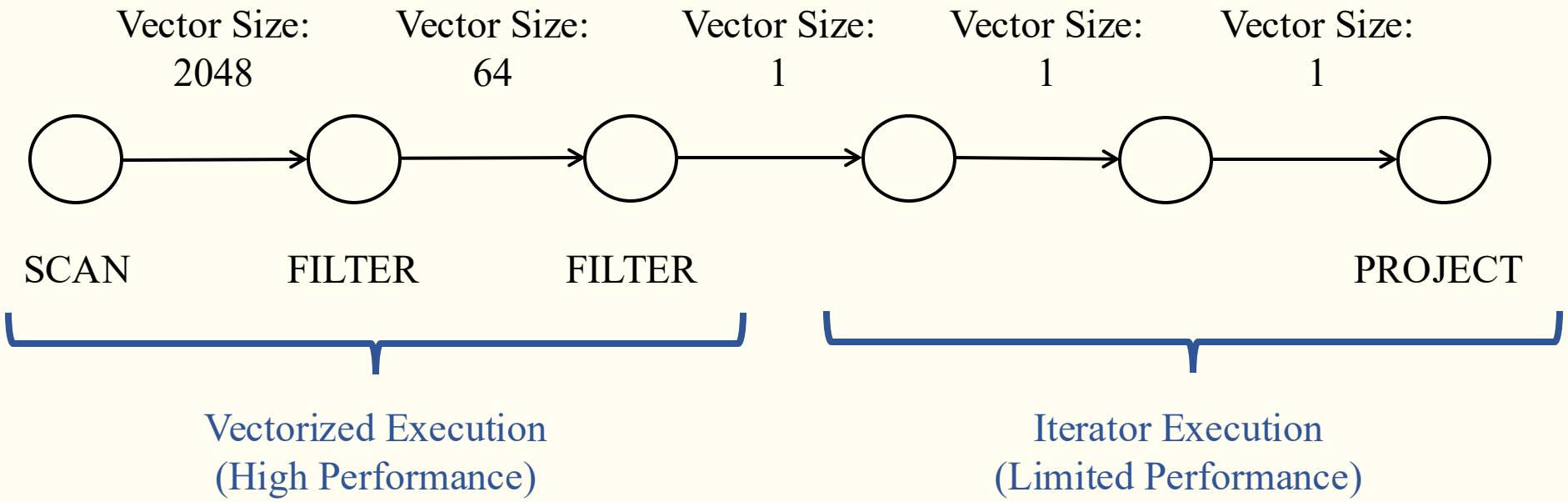


Buying Beer for a Party. An Accident.



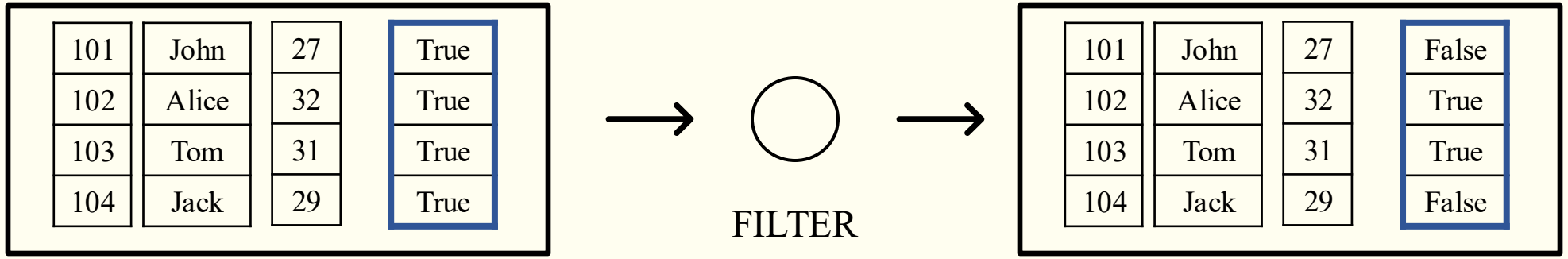
We take two crates from the shop
but only put one bottle in the fridge!

Small Chunk Problem

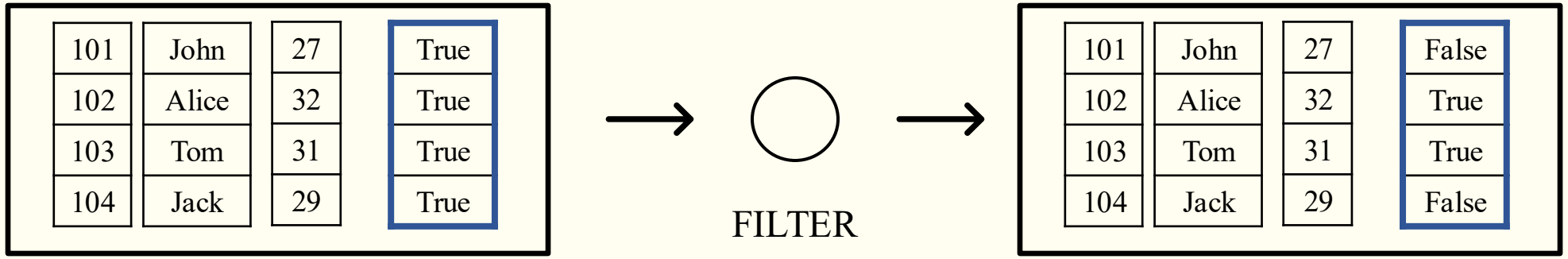


Chunk Size (and Data Volume) is Greatly Reduced During Execution

Chunk-reducing Operators: Filter

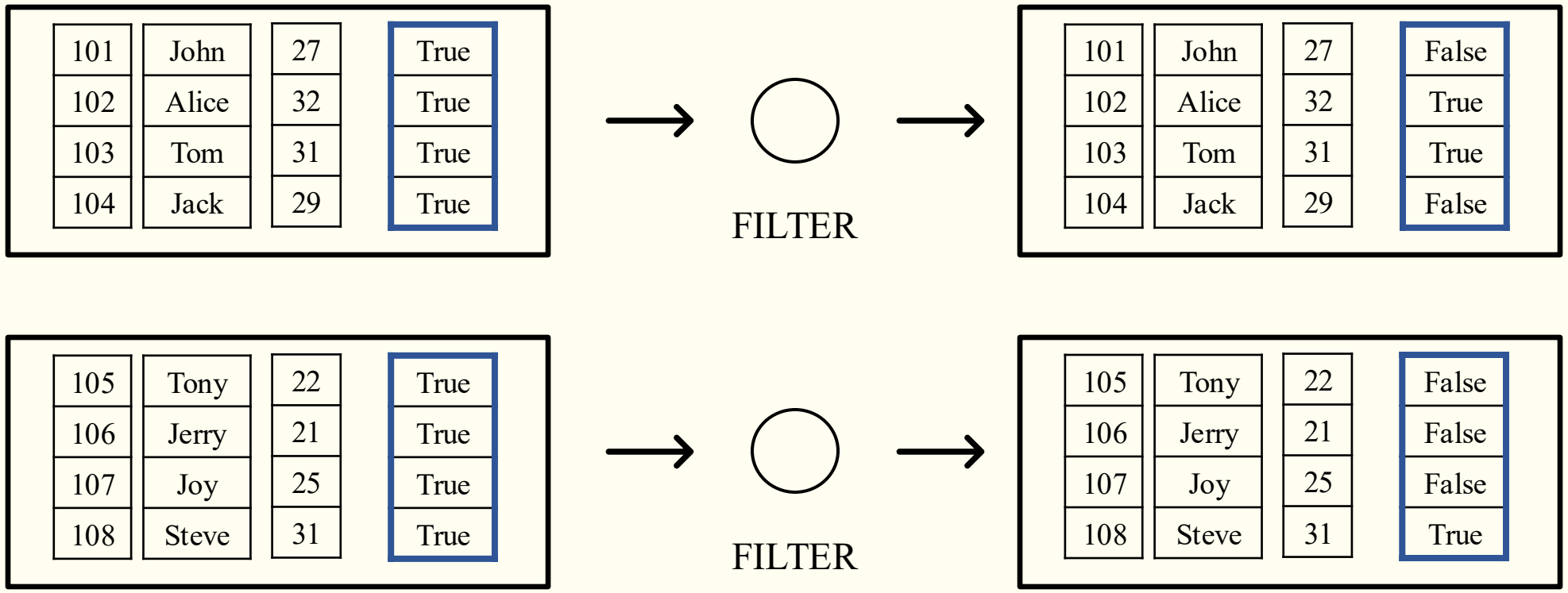


Chunk-reducing Operators: Filter

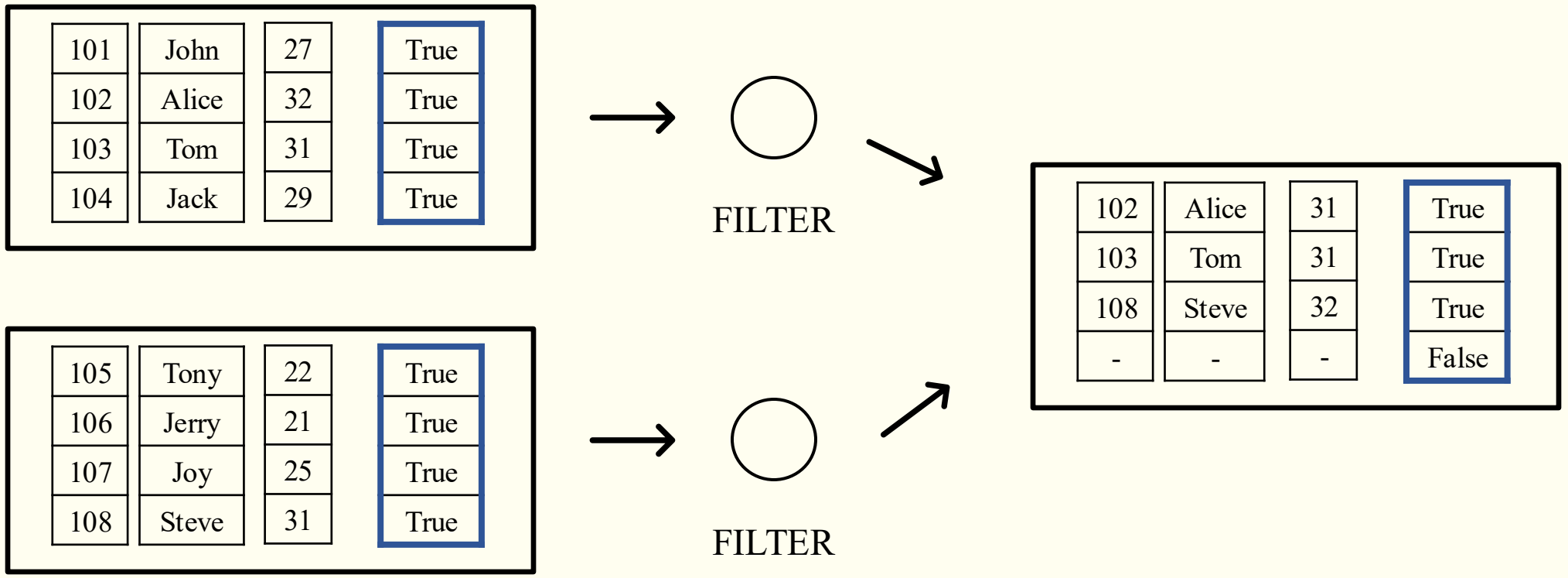


- Data Chunk Structure: Some Data Vectors + One Selection Vector
- Zero-copy Benefit

Chunk-reducing Operators: Filter



Chunk-reducing Operators: Filter

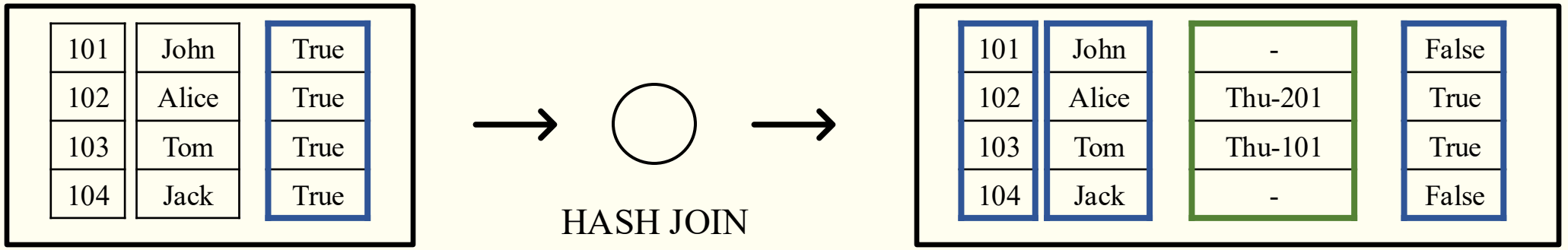


If we compact them, we have additional memory copies; otherwise, the vectorized execution suffers.

Chunk-reducing Operators: Hash Join

SELECT id, name, course_id,

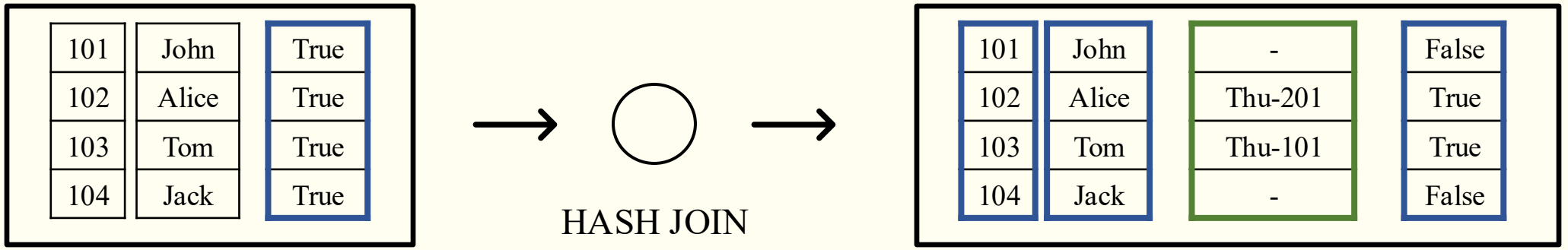
FROM students, courses



Chunk-reducing Operators: Hash Join

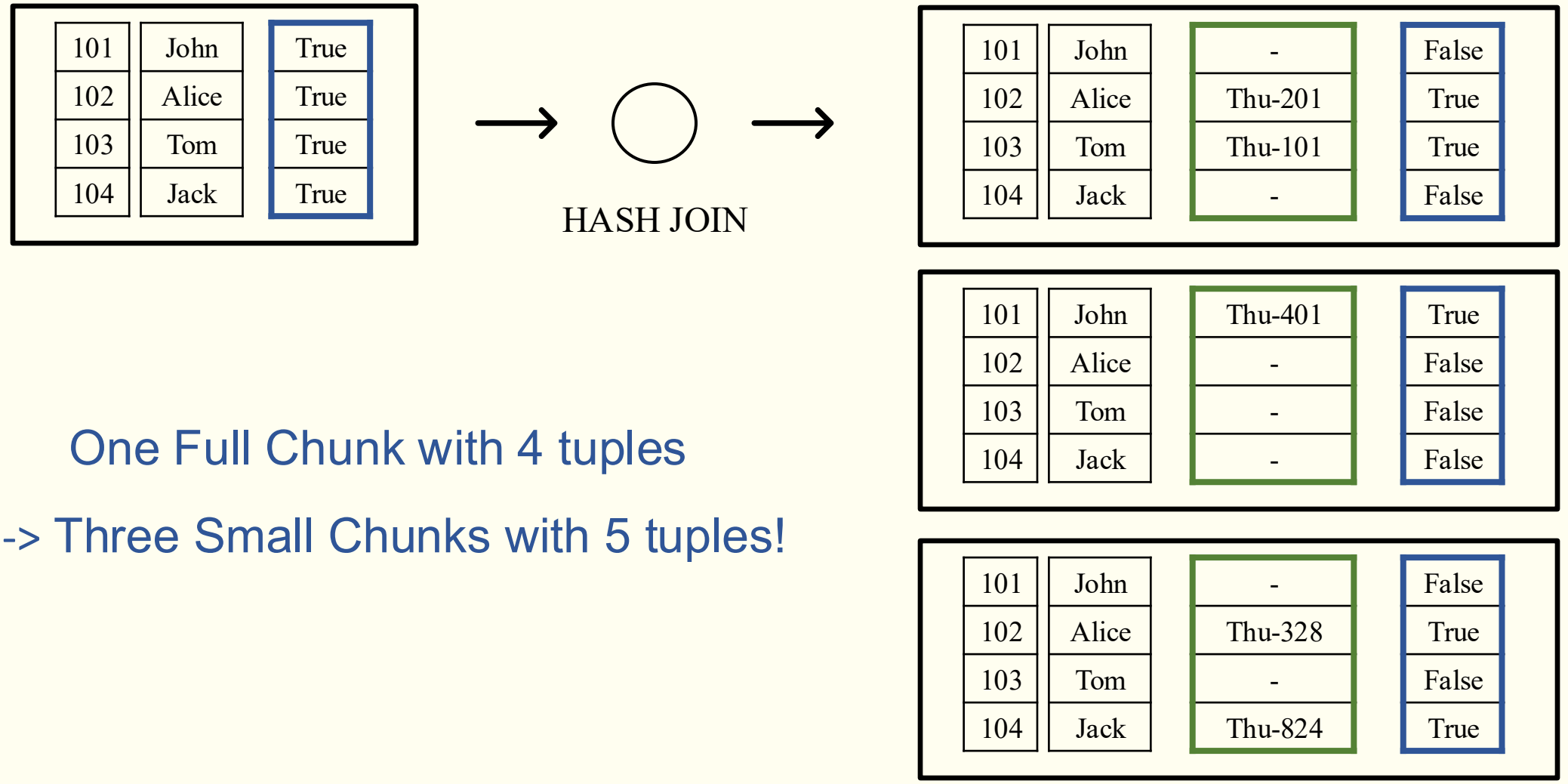
SELECT id, name, course_id,

FROM students, courses



- Zero-copy Benefit: Reuse the data in the column id and name.
- Copy the New Vector Values into the Result Chunk.

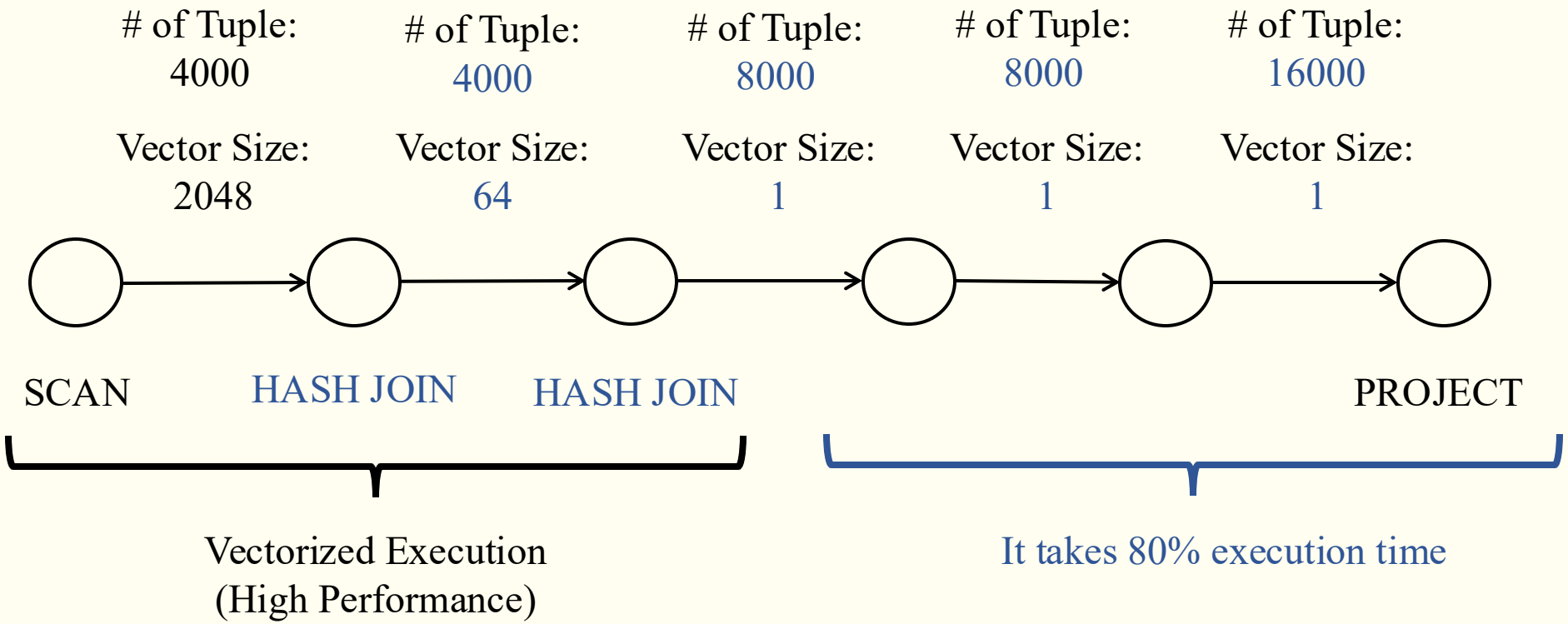
Chunk-reducing Operators: Hash Join



One Full Chunk with 4 tuples

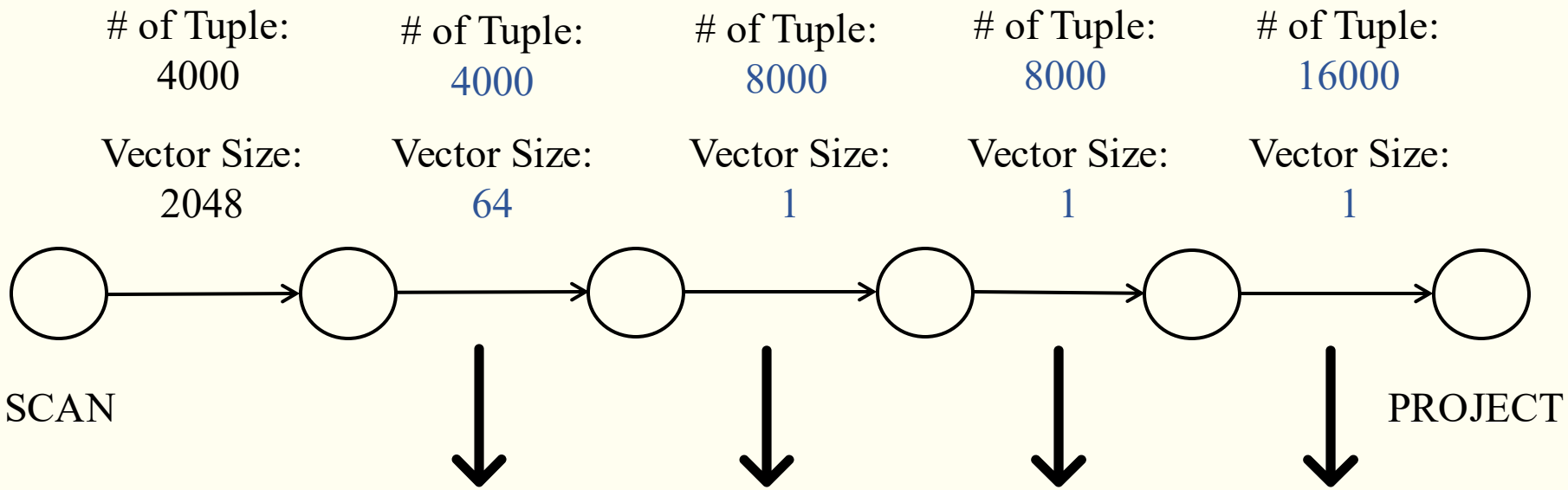
-> Three Small Chunks with 5 tuples!

Small Chunk Problem



Chunk Size is Greatly Reduced During Execution, but Data Volume is not Necessarily Reduced.

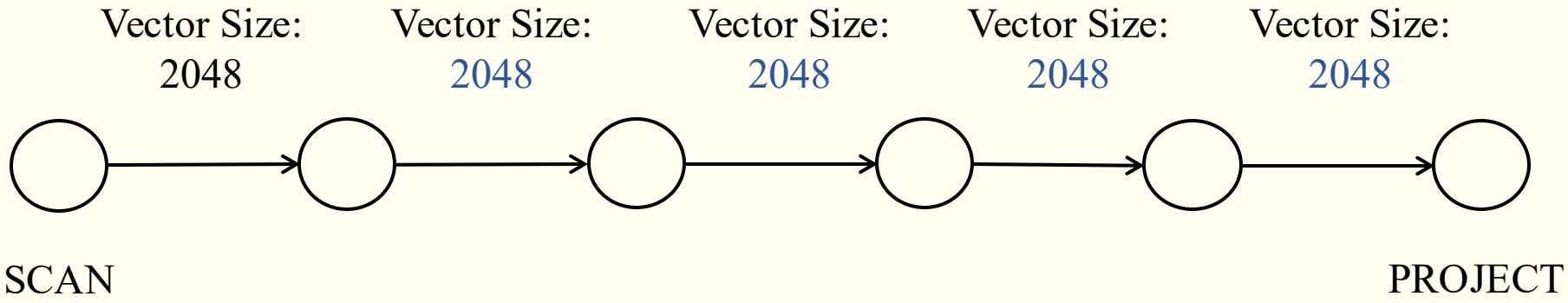
Small Chunk Problem



Given a Set of Data Chunks, Determine How to Compact Them to Minimize the Total Execution Time (= compaction time + compute time).

Compact as More as Possible

- **Full Compaction**: compacts all chunks containing less than 2048 tuples.

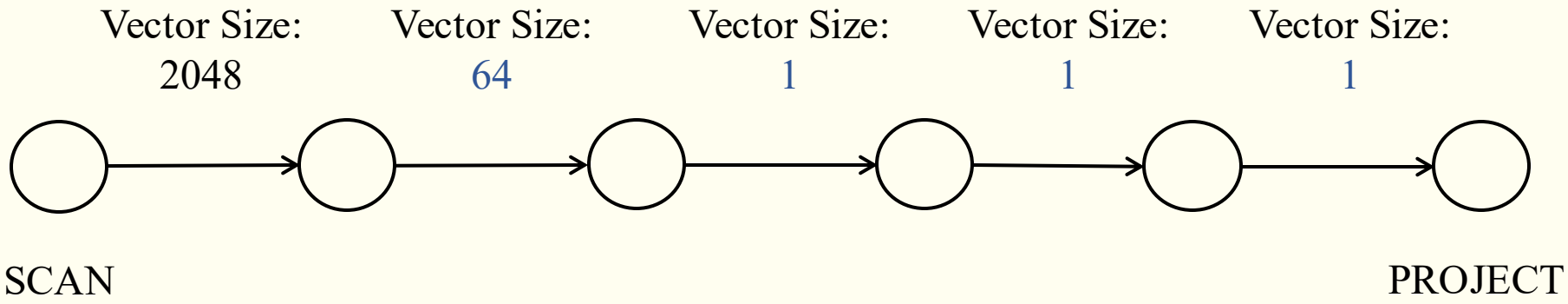


Too Many Memory Copies!

Consider the Case Where a Tuple is in the Length of 1 MB…

Compact as Less as Possible

- **No Compaction**: no chunks are compacted



Too Many Small Chunks!

Consider the Case Where a Tuple is in the Length of 10 Bytes…

Compact Small Chunks Only

- **Binary Compaction**: chunks smaller than a predefined threshold are compacted.
- A Trade-off Between Interpretation Overhead vs. Memory Copy
  - The smaller the chunk, the greater the benefit from such compaction.

Choosing a Predefined Threshold is Difficult

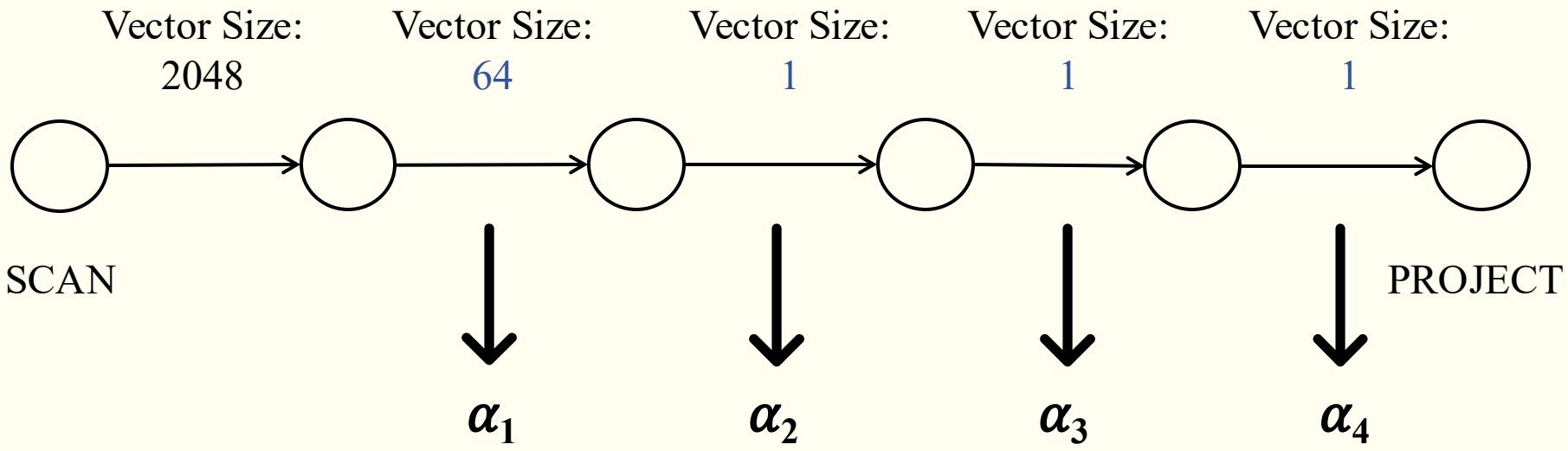
Because it involves Database Design and Workload Characteristics.

Our Solution 1: Learning Compaction

- **Learning Compaction**: chunks smaller than a Learned threshold are compacted.
- Morsel-driven Parallelism: data is divided into chunks, with each thread responsible for fetching and processing a chunk through the entire pipeline before moving on to the next.
  - Then, each chunk can serve as a sample for a learning algorithm
- Multi-armed Bandit Problem: select the optimal thresholds

Learning Compaction

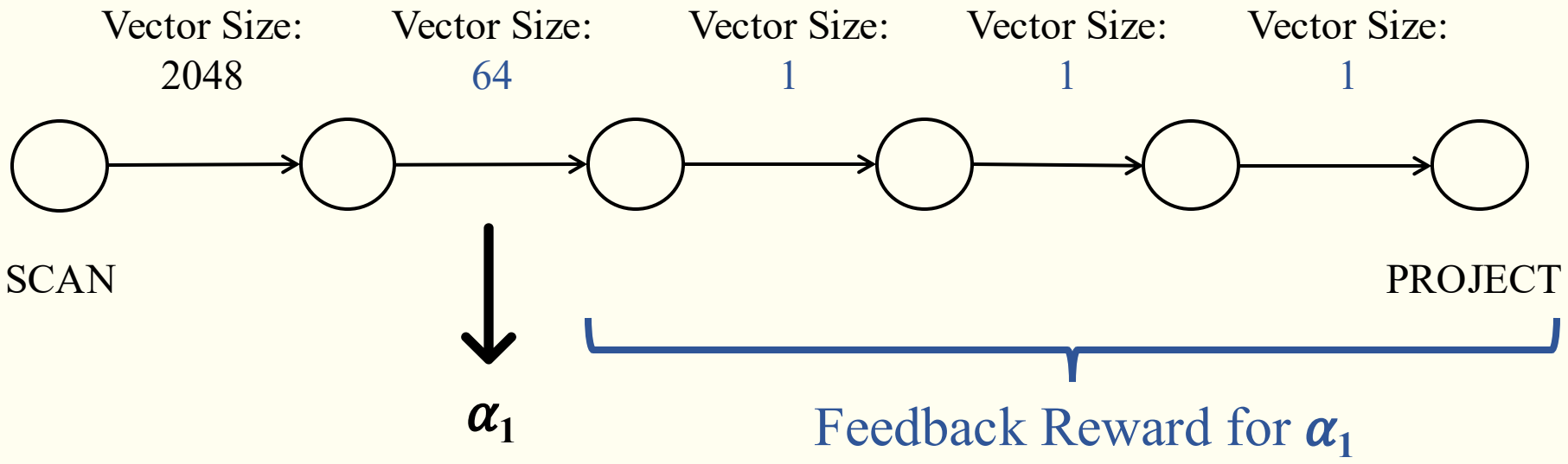
- **Learning Compaction**: chunks smaller than a learned threshold are compacted.



- Multi-armed Bandit Problem: select each alpha from 0, 32, 64, 128, …, 2048

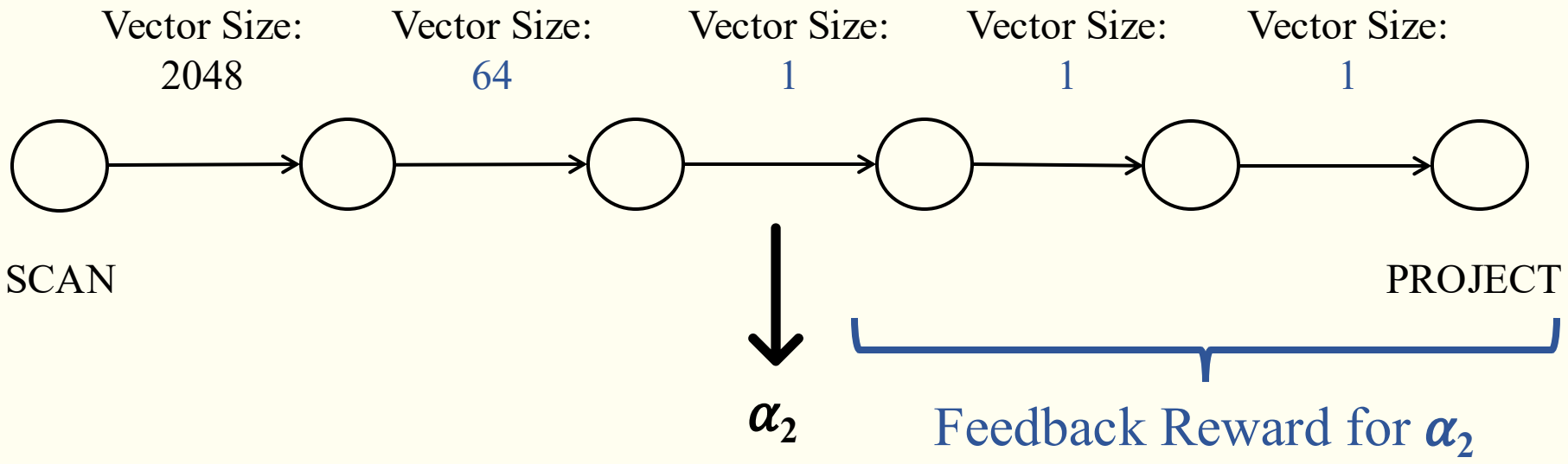
Learning Compaction

- **Learning Compaction**: chunks smaller than a learned threshold are compacted.



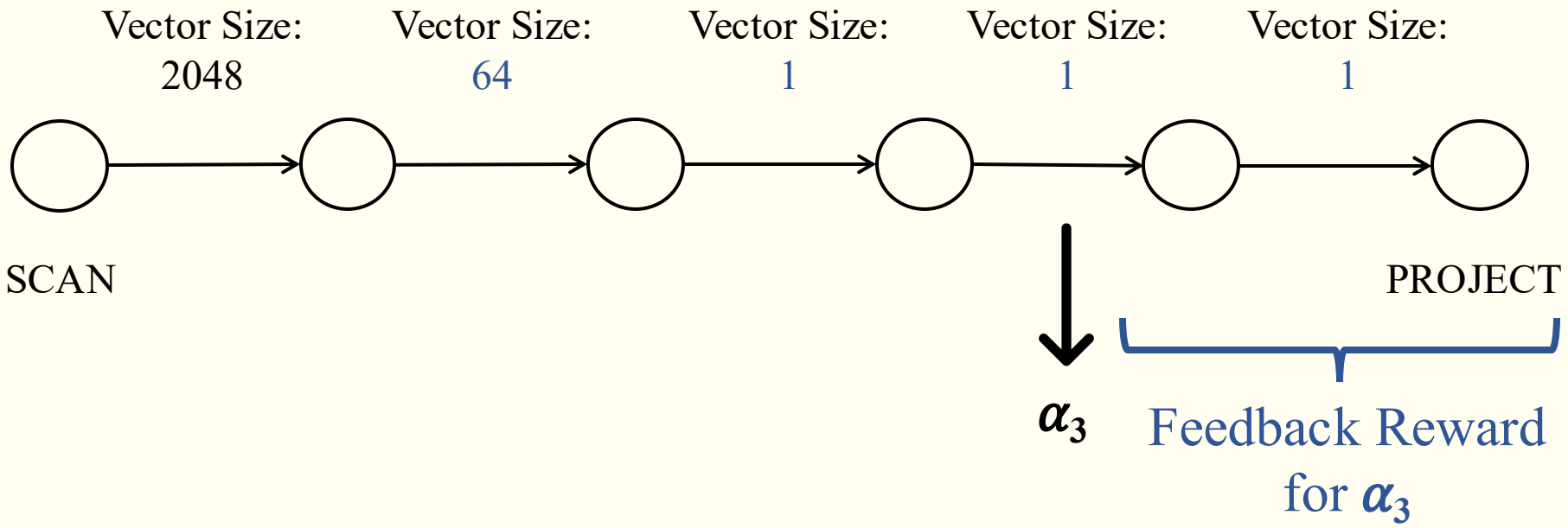
Learning Compaction

- **Learning Compaction**: chunks smaller than a learned threshold are compacted.

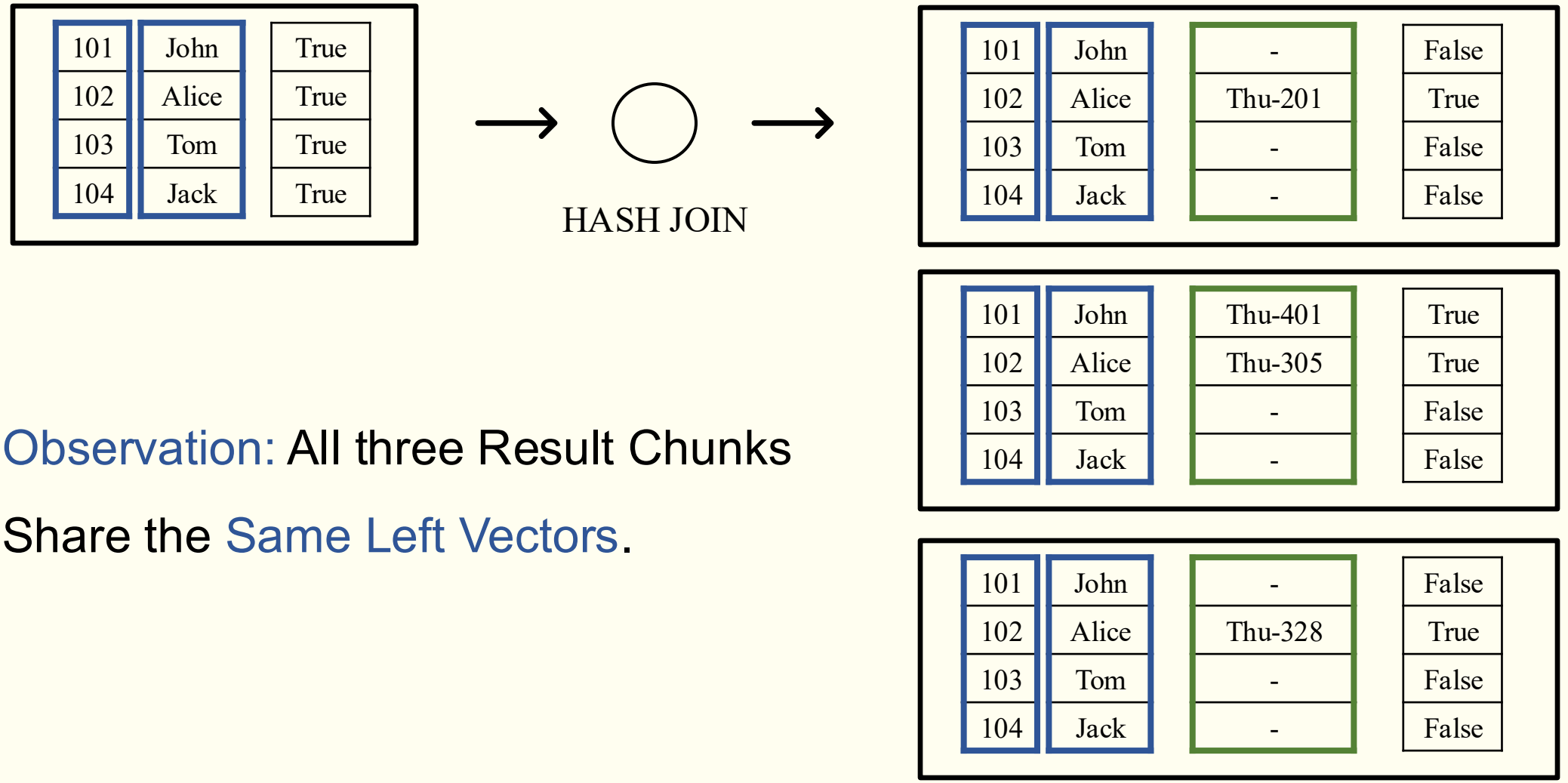


Learning Compaction

- **Learning Compaction**: chunks smaller than a learned threshold are compacted.



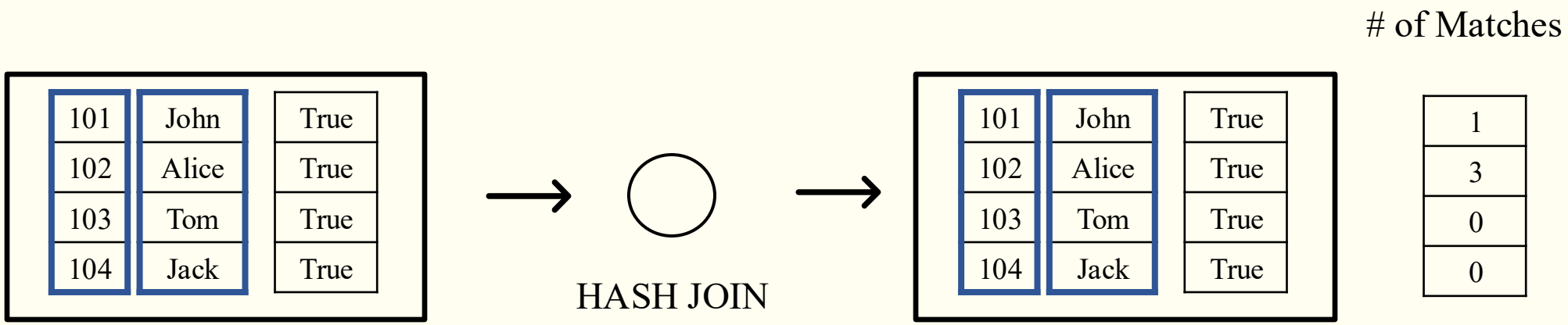
Our Solution 2: Logical Compaction



Observation: All three Result Chunks

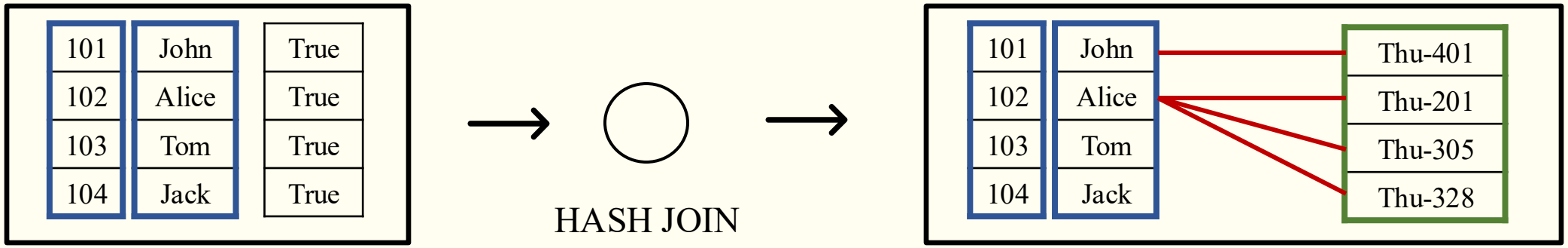
Share the Same Left Vectors.

Logical Compaction



Putting all generated tuples in one chunk seems to be a good idea, but how to keep the zero-copy property of the Left Side Vectors…

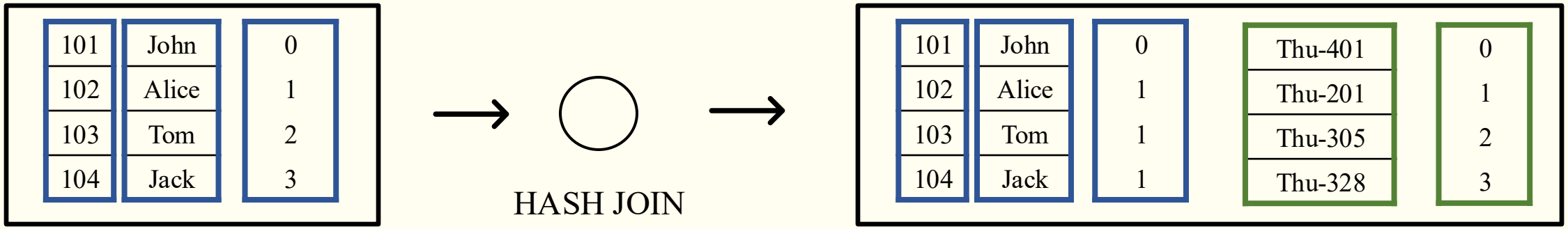
Logical Compaction



Requirements:

- Left-side vectors should be zero-copied.
- The mapping relationship between the left-side and right-side vectors should be maintained.
- The resulting chunk can be an input chunk to other operators.

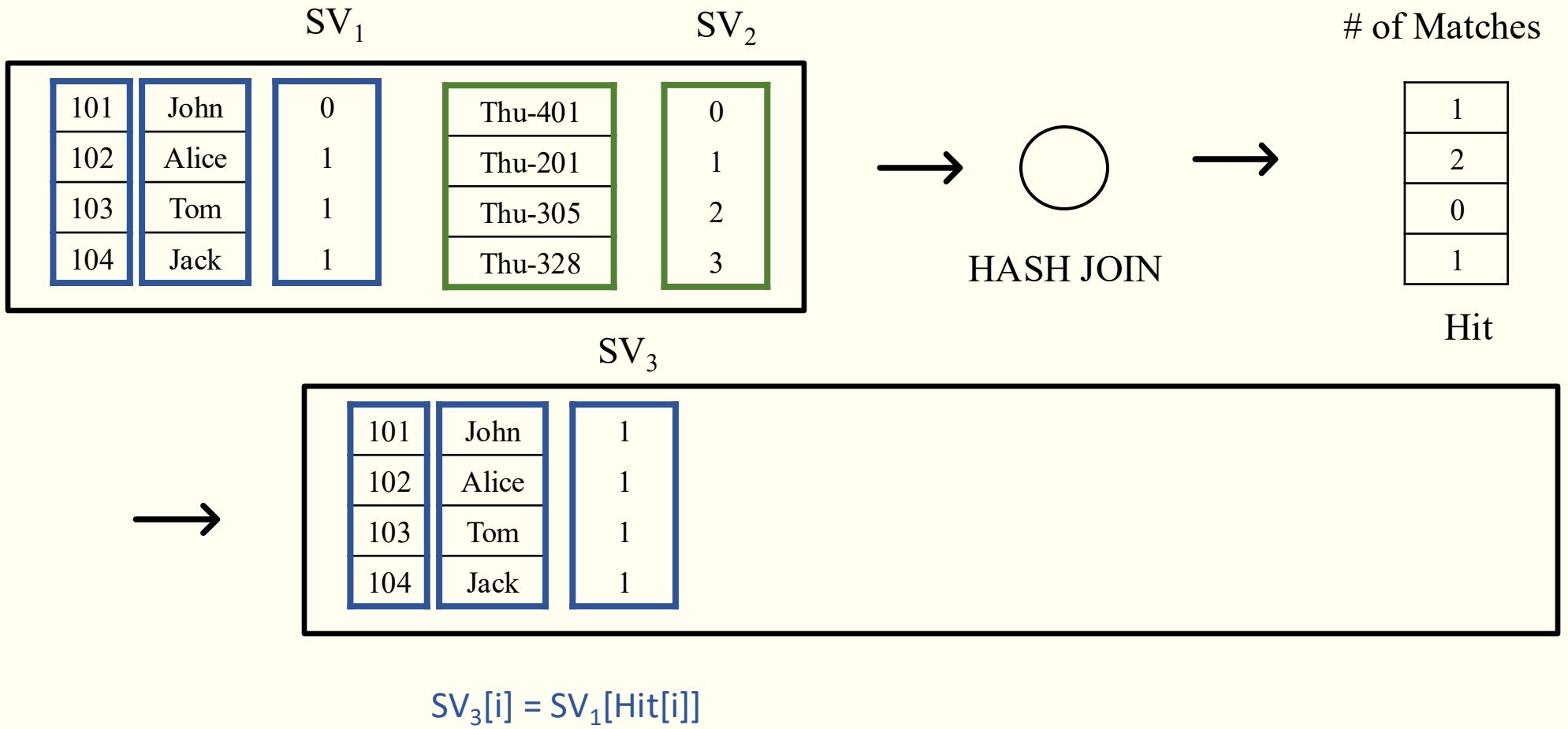
Our Solution 2: Logical Compaction



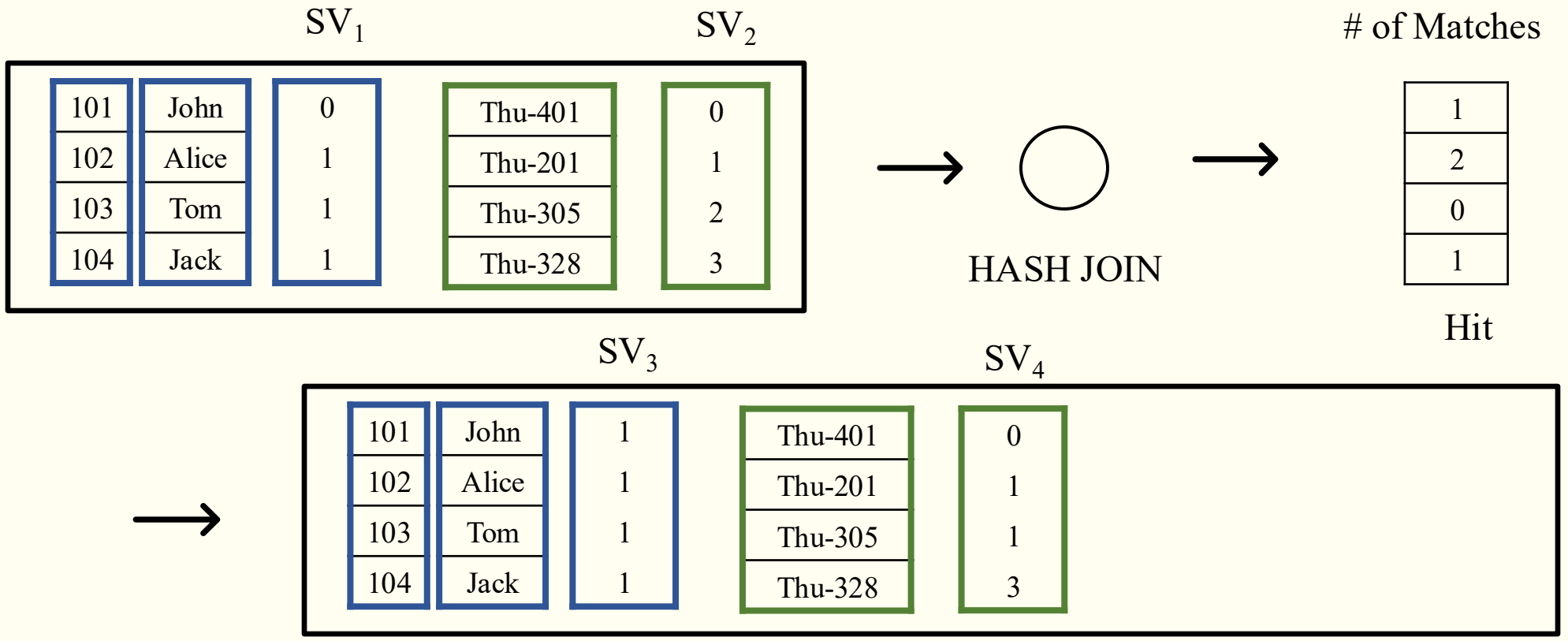
Our Solution: Selection Vector (SV) with Repeated Values:

- Left-side vectors have an SV.
- Right-side vectors have another SV.

Hash Probe the Result Chunk Again

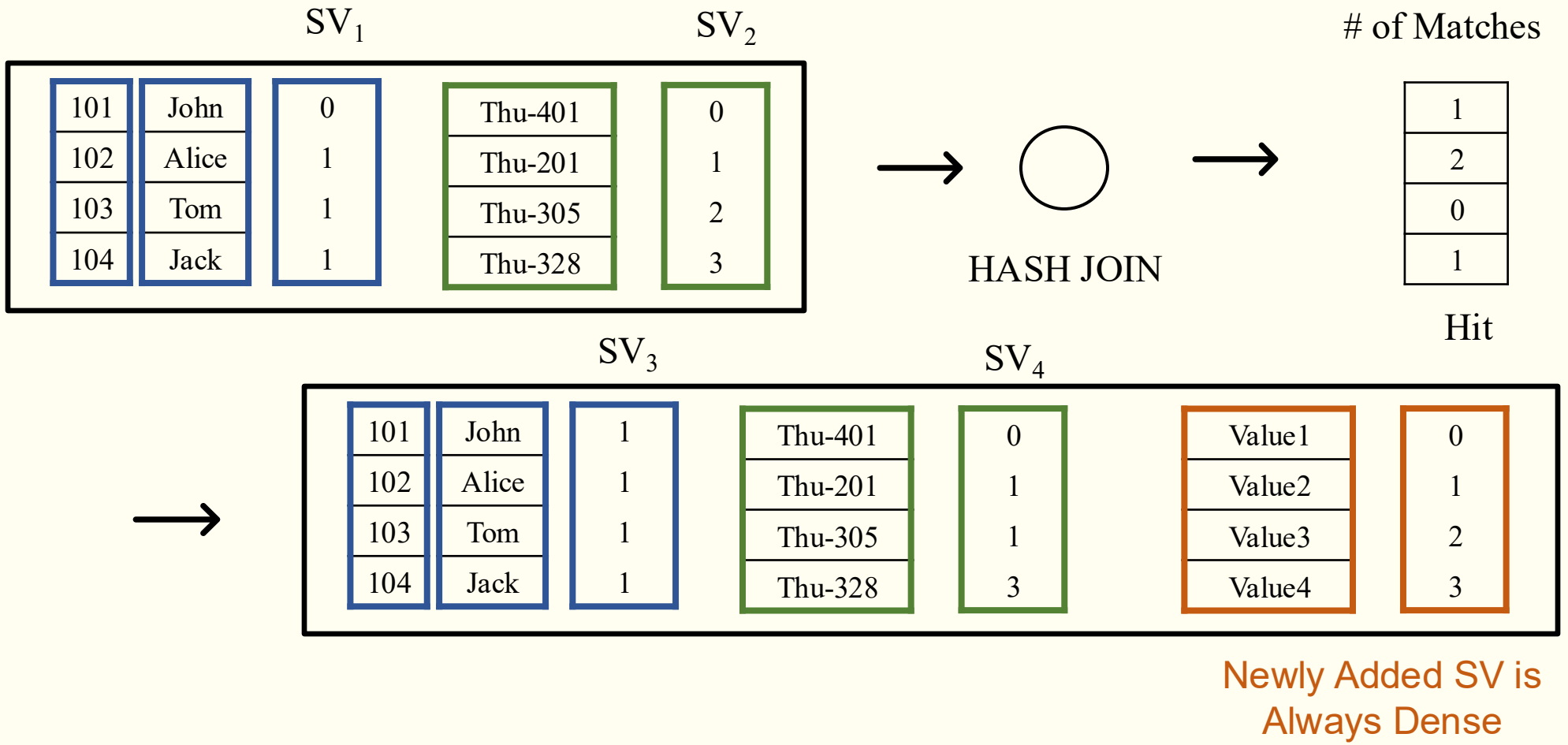


Hash Probe the Result Chunk Again

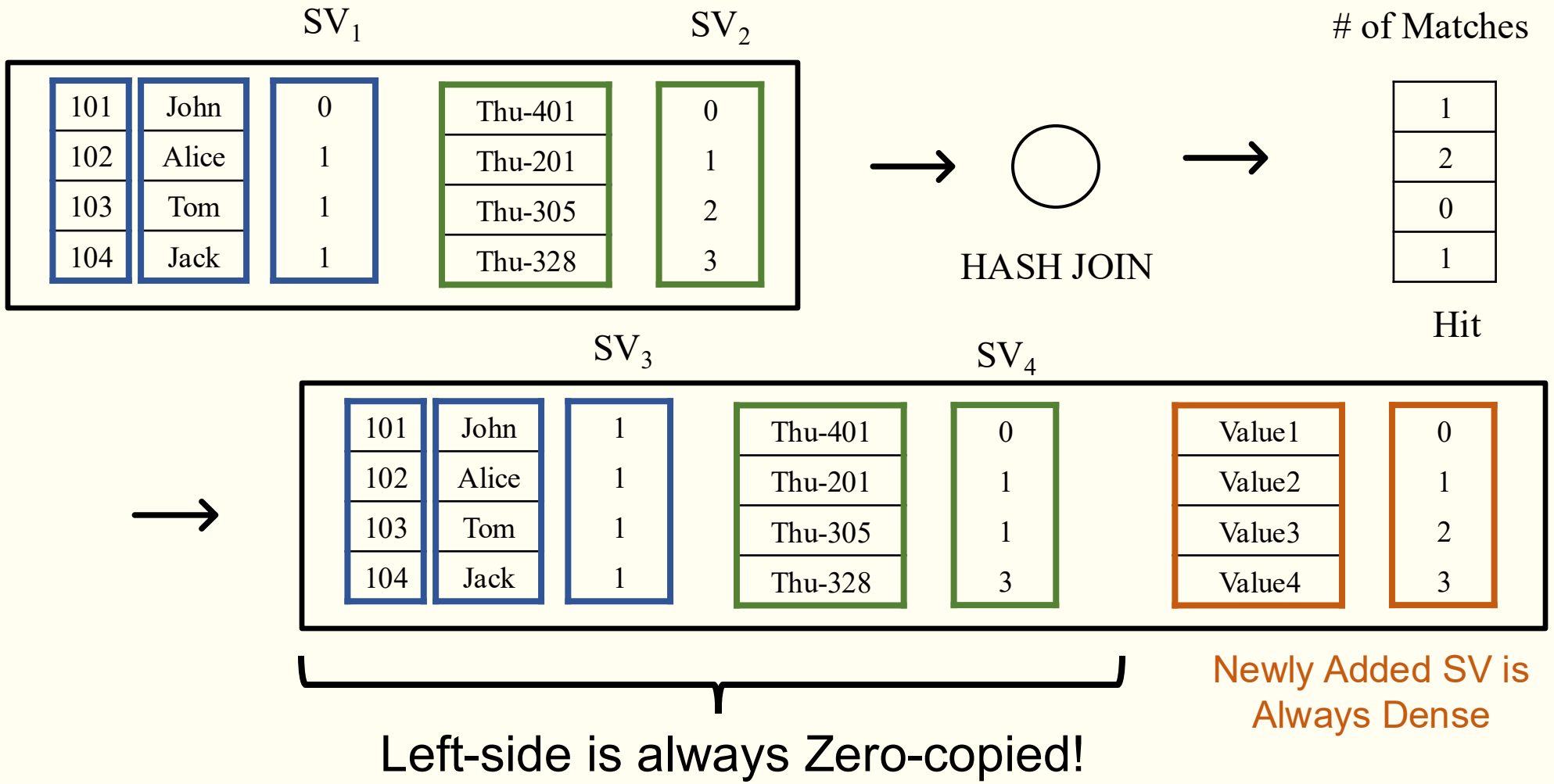


$SV_4[i] = SV_2[Hit[i]]$

Hash Probe the Result Chunk Again



Hash Probe the Result Chunk Again



End-to-End Performance in DuckDB

- We integrate our solutions into DuckDB and measure the end-to-end performance.
- Smart Compaction: Learning Compaction + Logical Compaction

End-to-End Performance in DuckDB

- We integrate our solutions into DuckDB and measure the end-to-end performance.
- Smart Compaction: Learning Compaction + Logical Compaction