Course code : **CSE3009**
Course title : **No SQL Data Bases**
Module : **5**
Topic : **4**

Column-store internals

Objectives

This session will give the knowledge about

- Late Materialization
- Joins

Late Materialization

In a particular query plan, various tables are accessed. The process of accessing several columns to create a record is known as tuple creation.

Creation of tuples based on the information present in the query plan is known as materialization.

In a query plan tuple is created if the join condition is satisfied.

The process where the tuples (where join condition is satisfied) are stitched together as soon as possible during a query plan is known as Early materialization

Late Materialization

The process where the tuples are not created until some part of the query plan is processed (apart from the checking of the join condition) is termed as Late Materialization.

It generally is the operation specified in the WHERE condition of a query.

This process is useful because it saves the CPU from performing operation of joining the unnecessary tuples

Late Materialization

Select sum(R.a) from R, S where R.c = S.b and 5<R.a<20 and 40<R.b<50 and 30<S.a<40

Initial Status

Relation R

Ra	Rb	Rc
3	12	12
16	34	34
56	75	53
9	45	23
11	49	78
27	58	65
8	97	33
41	75	21
19	42	29
35	55	0

Relation S

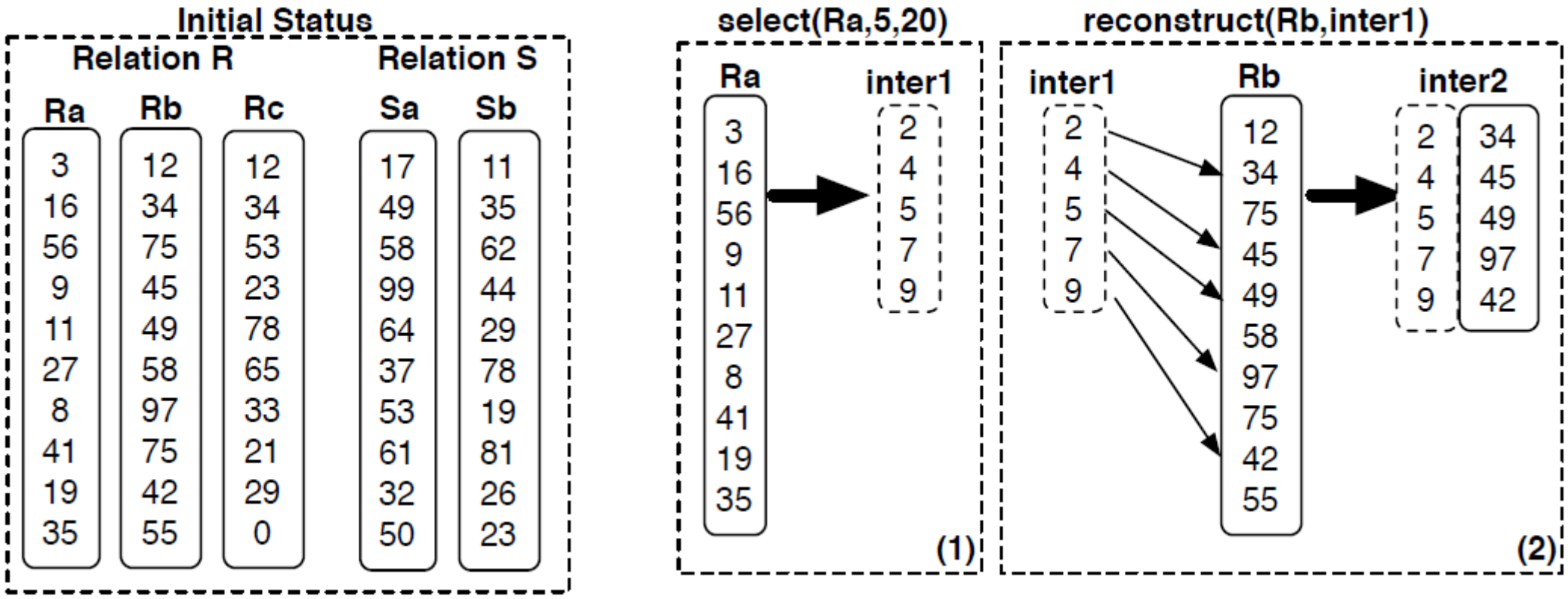
Sa	Sb
17	11
49	35
58	62
99	44
64	29
37	78
53	19
61	81
32	26
50	23

Query and Query Plan (MAL Algebra)

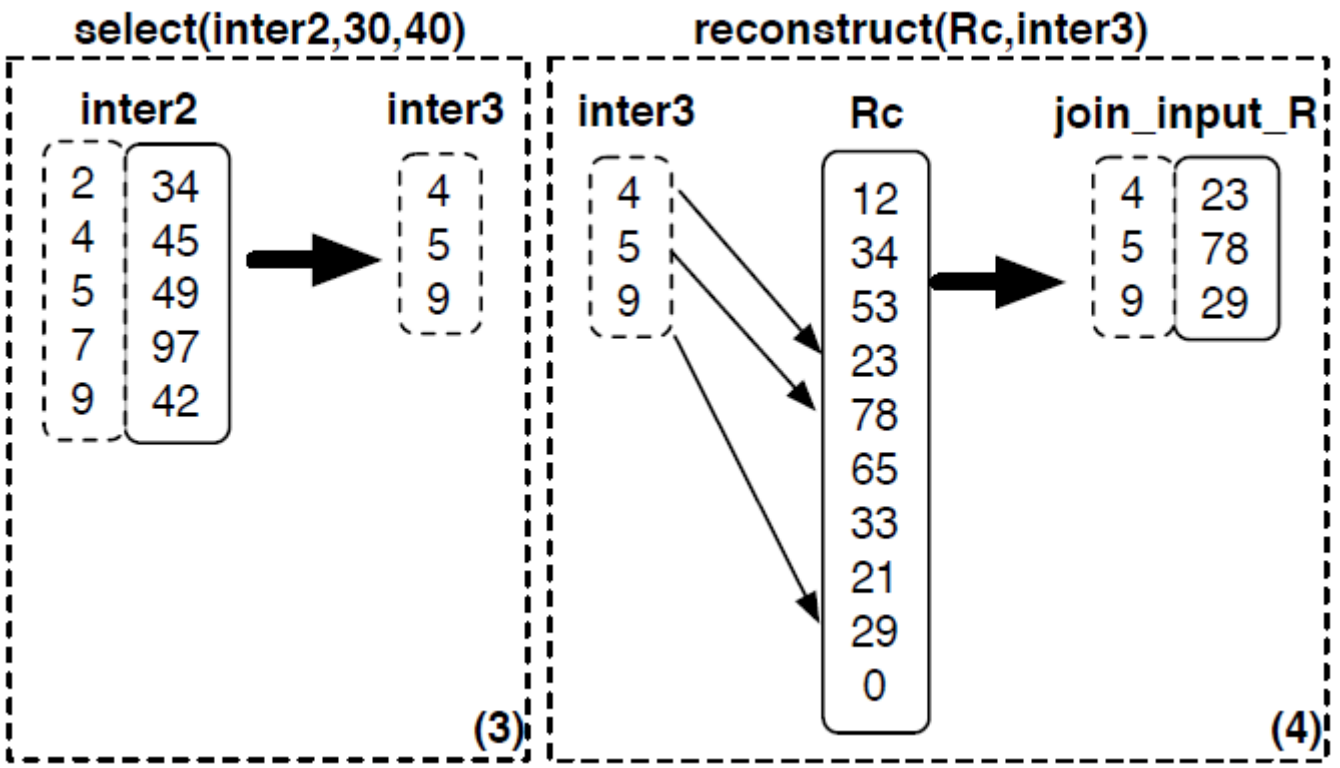
select sum(R.a) from R, S where R.c = S.b and 5<R.a<20 and 40<R.b<50 and 30<S.a<40

1. inter1 = **select**(Ra,5,20)
2. inter2 = **reconstruct**(Rb,inter1)
3. inter3 = **select**(inter2,30,40)
4. join_input_R = **reconstruct**(Rc,inter3)
5. inter4 = **select**(Sa,55,65)
6. inter5 = **reconstruct**(Sb,inter4)
7. join_input_S = **reverse**(inter5)
8. join_res_R_S = **join**(join_input_R,join_input_S)
9. inter6 = **voidTail**(join_res_R_S)
10. inter7 = **reconstruct**(Ra,inter6)
11. result = **sum**(inter7)

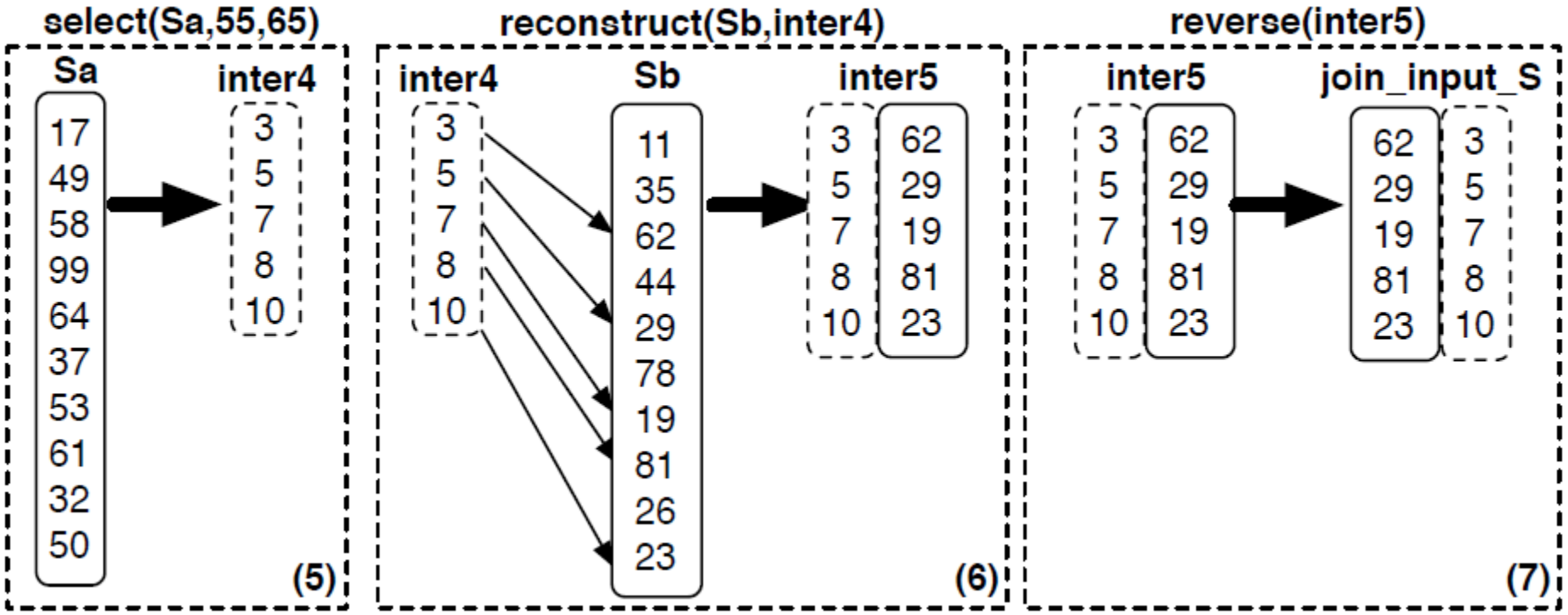
Late Materialization



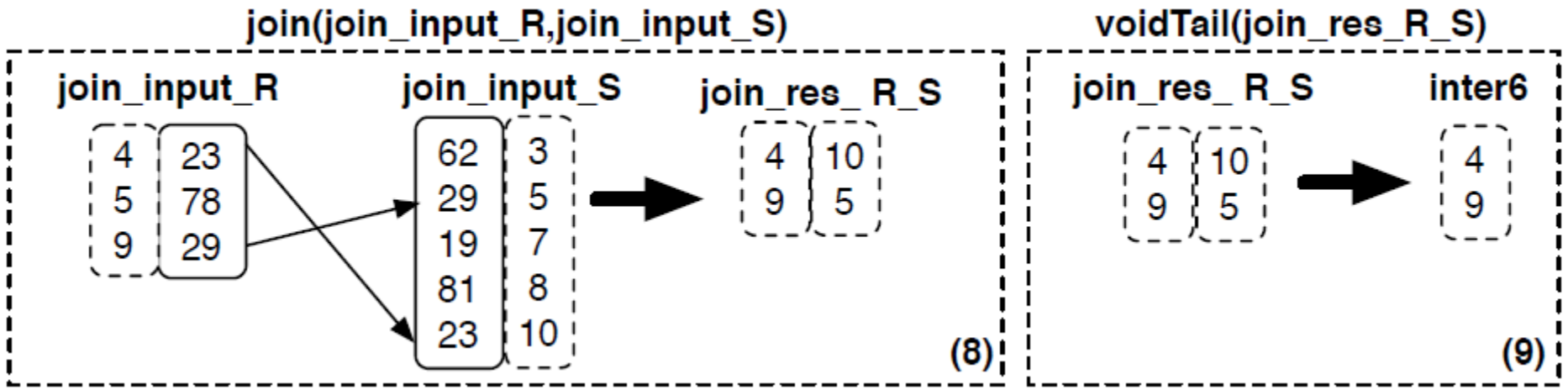
Late Materialization



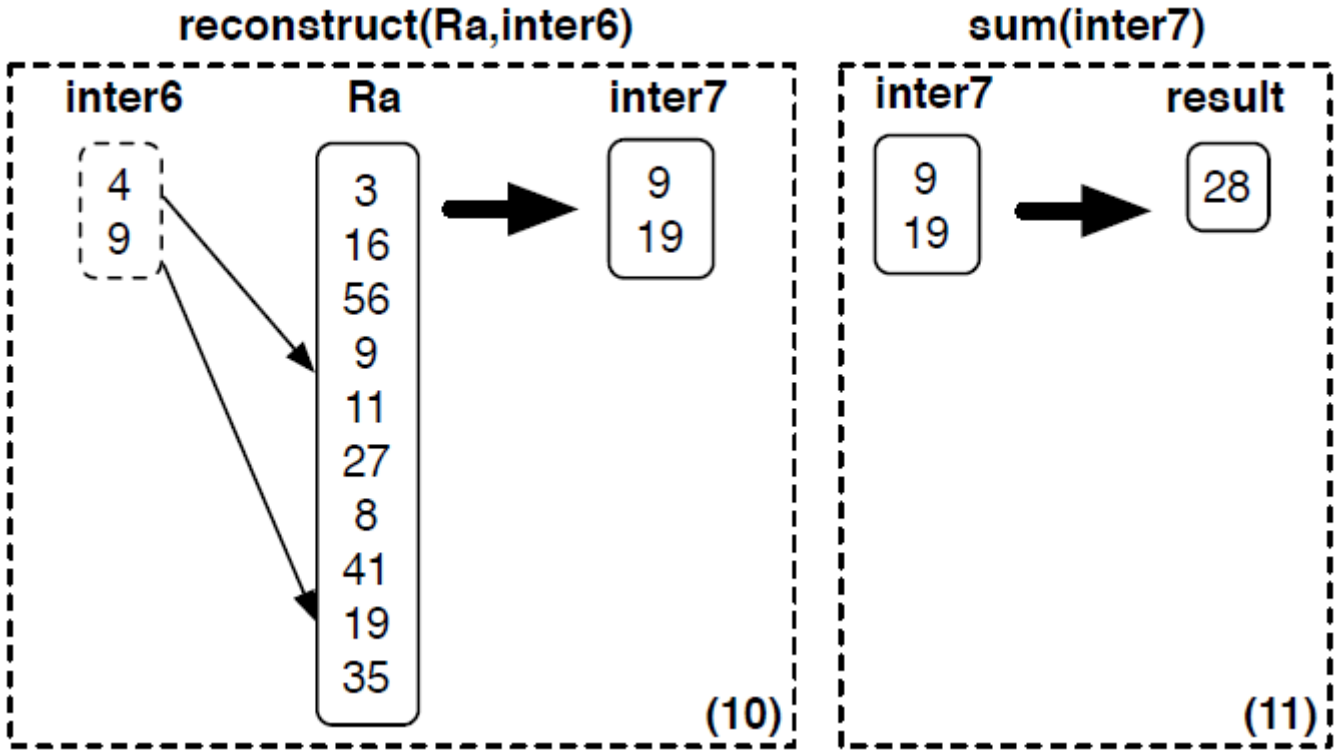
Late Materialization



Late Materialization



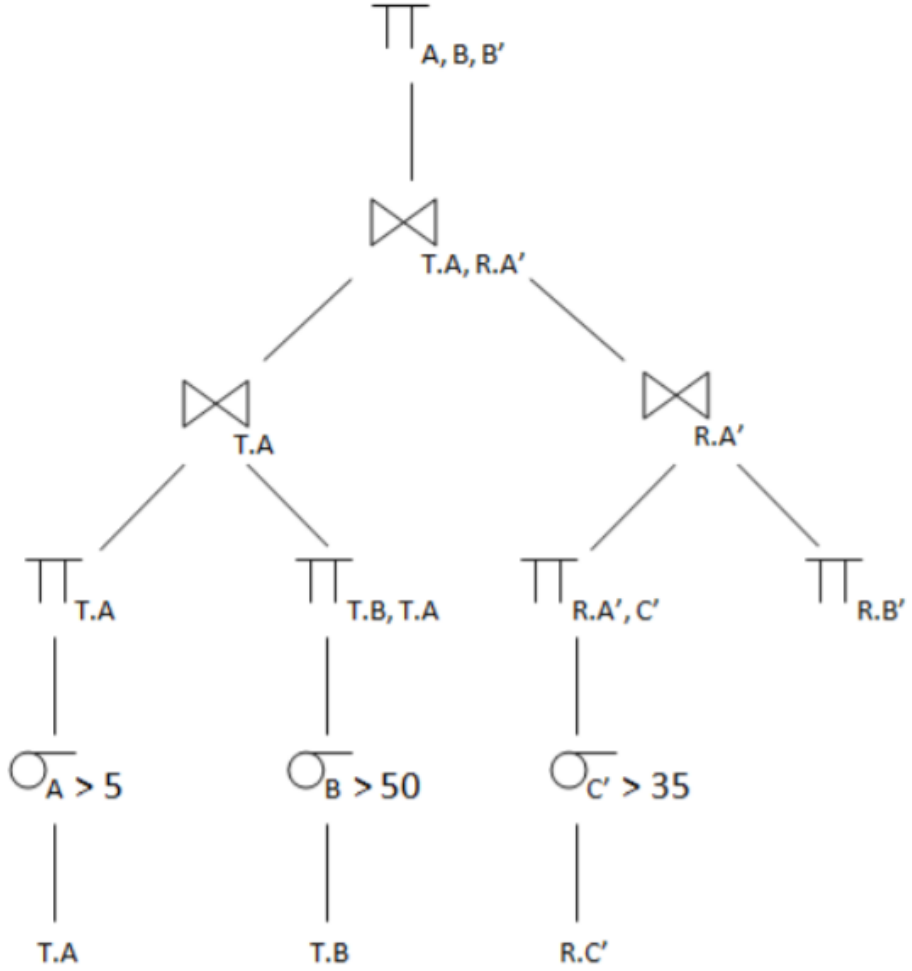
Late Materialization



Late Materialization

Executing the following query which includes two table T,R

Query 1:SELECT T.A,T.B,R.B’ FROM T JOIN R ON (T.A=R.A’) WHERE A>5 AND B<50 AND C’<35



Joins

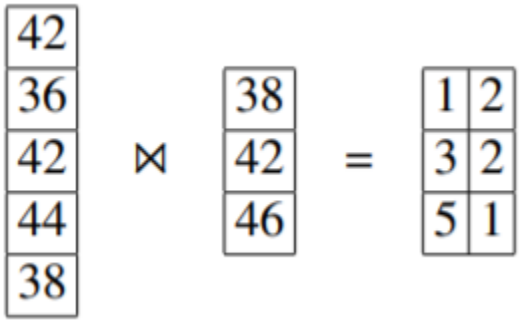
If an early materialization strategy is used relative to a join, tuples have already been constructed before reaching the join operator, so the join functions as it would in a standard row-store system and outputs tuples.

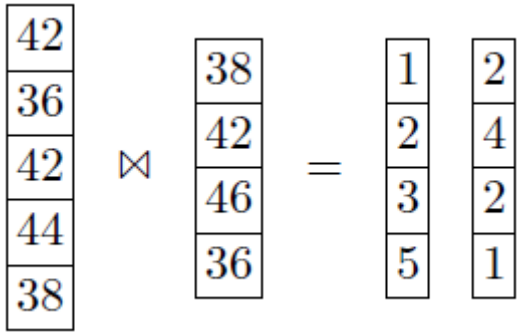
An alternative algorithm can be used with a late materialization strategy, however. In this case, only the columns that compose the join predicate are input to the join.

The output of the join is a set of pairs of positions in the two input relations for which the predicate succeeded.

Joins

Example:





For many join algorithms, the output positions for the left (outer) input relation will be sorted while the output positions of the right (inner) input relation will not.

This is because the positions in the left column are usually iterated through in order, while the right relation is probed for join predicate matches.

Joins

Consider,

SELECT emp.age, dept.name FROM emp, dept WHERE emp.dept_id = dept.id

Unsorted positional output is problematic since typically after the join, other columns from the joined tables will be needed.

Unordered positional lookups are problematic since extracting values from a column in this unordered fashion requires jumping around storage for each position, causing significant slowdown.

Jive join

When we joined the column of size with a column of size 4 above, we received the following positional output:

1	2
2	4
3	2
5	1

The list of positions for the right (inner) table is out of order.

Let’s assume that we want to extract the customer name attribute from the inner table according to this list of positions, which contains the following four customers.

Jive join

The basic idea of the Jive join is to add an additional column to the list of positions that we want to extract, that is a densely increasing sequence of integers:

1	2
2	4
3	2
5	1

Smith
Johnson
Williams
Jones

2	1
4	2
2	3
1	4

1	4
2	1
2	3
4	2

1	4	Smith
2	1	Johnson
2	3	Johnson
4	2	Jones

2	1	Johnson
4	2	Jones
2	3	Johnson
1	4	Smith

Group-By

Group-by is typically a hash-table based operation in modern column-stores and thus it exploits similar properties as discussed in the join section.

In particular, we may create a compact hash table, i.e., where only the grouped attribute can be used, leading in better access patterns when probing.

Aggregation

Aggregation operations make heavy use of the columnar layout.

In particular, they can work on only the relevant column with tight for-loops.

For example, assume sum(), min(), max(), avg() operators; such an operator only needs to scan the relevant column (or intermediate result which is also in a columnar form), maximizing the utilization of memory bandwidth.

An example is shown in Figure at Slide 5 in Step 11, where we see that the sum operator may access only the relevant data in a columnar form.

Arithmetic Operations

Other operators that may be used in the select clause in an SQL query, i.e., math operators (such as +,-*,/) also exploit the columnar layout to perform those actions efficiently.

However, in these cases, because such operators typically need to operate on groups of columns,

e.g., select A+B+C From R ...,

they typically have to materialize intermediate results for each action.

Inserts/updates/deletes

A natural approach to implement the write-store is to store differences (inserts, deletes, and updates) in an in-memory structure.

MonetDB uses plain columns, i.e., for every base column in the schema there are two auxiliary columns to store pending inserts and pending deletes; an update is a delete followed by an insert.

C-Store proposed that the write optimized store could also use a row-format which speeds up updates even more as only one I/O is needed to write a single new row (but merging of updates in the column format becomes potentially more expensive).

Inserts/updates/deletes

The VectorWise system uses a novel data structure, called Positional Delta Trees (PDTs) to store differences.

The key advantage is that merging is based on knowledge of the position where differences apply, and not on the sort key of the table, which can be composite and complex.

Summary

This session will give the knowledge about

- Late Materialization
- Joins