<u>A</u>dvanced <u>D</u>ata <u>M</u>anagement for Data Analysis

Stefan Manegold

Data Management @ LIACS

Group leader Database Architectures
Centrum Wiskunde & Informatica (CWI)
Amsterdam

s.manegold@liacs.leidenuniv.nl
http://www.cwi.nl/~manegold/

ADM: Agenda

- 07.09.2022: Lecture 1: **Introduction**
- 14.09.2022: Lecture 2: **SQL Recap**

  (plus Assignment 1 [in groups; 3 weeks]: TPC-H benchmark)
- 21.09.2022: Lecture 3: **Column-Oriented Database Systems (1/6) - Motivation & Basic Concepts**
- 28.09.2022: Lecture 4: **Column-Oriented Database Systems (2a/6) - Selected Execution Techniques (1/2)**
- 05.10.2022: Lecture 5: **Column-Oriented Database Systems (2b/6) - Selected Execution Techniques (2/2)**

  (plus Assignment 2 [in groups; 3 weeks]: Compression techniques)
- 12.10.2022: Lecture 6: **Column-Oriented Database Systems (3/6) - Cache Conscious Joins**
- 19.10.2022: Lecture 7: **Column-Oriented Database Systems (4/6) - "Vectorized Execution"**
- ~~26.10.2022~~: ***No lecture!***
- 02.11.2022: Lecture 8: **DuckDB: An embedded database for data science (1/2) (guest lecture & *hands-on*)**

  (plus Assignment 3 [individual; 2 weeks]: Analysing NYC Cab dataset with DuckDB)
- 09.11.2022: Lecture 9: **DuckDB: An embedded database for data science (2/2) (guest lecture & *hands-on*)**
- 16.11.2022: Lecture 10: **Branch Misprediction & Predication**

  (plus Assignment 4 [individual; 2 weeks]: Predication)
- 23.11.2022: Lecture 11: **Column-Oriented Database Systems (5/6) - Adaptive Indexing**
- 30.11.2022: Lecture 12: **Column-Oriented Database Systems (6/6) - Progressive Indexing**

ADM: Literature

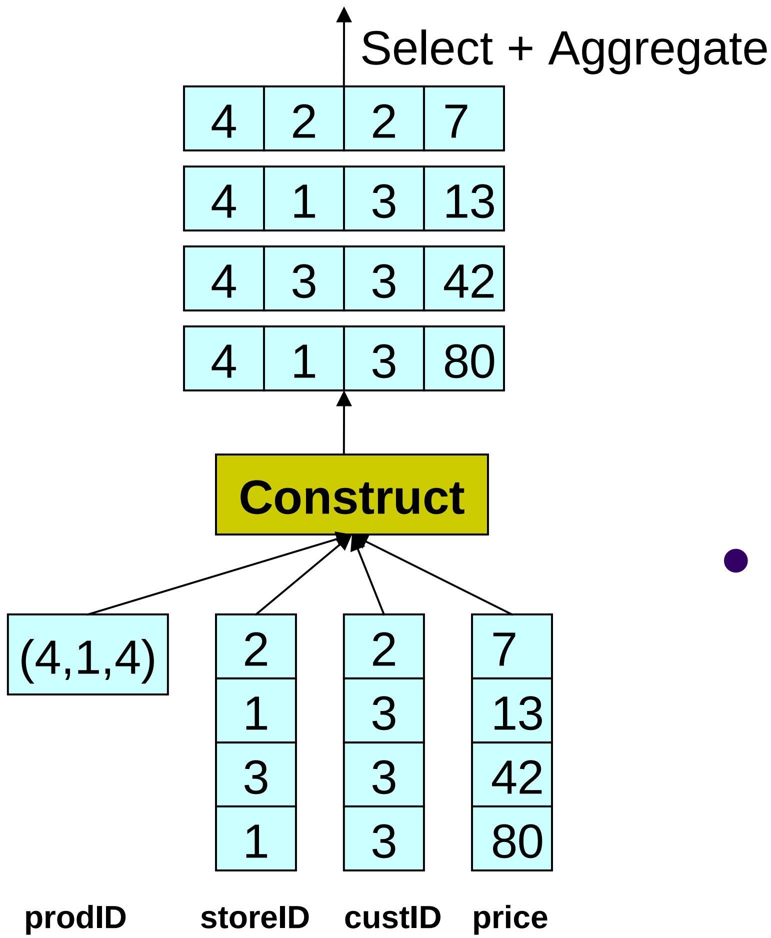
- **Column-Oriented Database Systems (2/6) - Selected Execution Techniques**
  - Compression
    - “Compressing Relations and Indexes”. Goldstein, Ramakrishnan, Shaft. ICDE’98.
    - “Query optimization in compressed database systems”. Chen, Gehrke, Korn. SIGMOD’01.
    - “Super-Scalar RAM-CPU Cache Compression”. Zukowski, Heman, Nes, Boncz. ICDE’06.
    - “Integrating Compression and Execution in Column-Oriented Database Systems”. Abadi, Madden, Ferreira. SIGMOD’06.
    - “Improved Word-Aligned Binary Compression for Text Indexing”. Ahn, Moffat. TKDE’06.
  - Tuple Materialization
    - “Materialization Strategies in a Column-Oriented DBMS”. Abadi, Myers, DeWitt, Madden. ICDE’07.
    - “Column-Stores vs Row-Stores: How Different are They Really?”. Abadi, Madden, Hachem. SIGMOD’08.
    - “Query Processing Techniques for Solid State Drives”. Tsirogiannis, Harizopoulos Shah, Wiener, Graefe. SIGMOD’09.
    - “Self-organizing tuple reconstruction in column-stores“. Idreos, Manegold, Kersten. SIGMOD’09.
  - Join
    - “Fast Joins using Join Indices”. Li and Ross. VLDBJ 8:1-24, 1999.

When should columns be projected?

- Where should column projection operators be placed in a query plan?
  - Row-store:
    - Column projection involves removing unneeded columns from tuples
    - Generally done as early as possible
  - Column-store:
    - Operation is almost completely opposite from a row-store
    - Column projection involves reading needed columns from storage and extracting values for a listed set of tuples
      - This process is called "materialization"
    - Early materialization (EM): project columns at beginning of query plan
      - Straightforward since there is a one-to-one mapping across columns
    - Late materialization (LM): wait as long as possible for projecting columns
      - More complicated since selection and join operators on one column obfuscates mapping to other columns from same table
    - Most column-stores construct tuples at column projection time
      - Many database interfaces expect output in regular tuples (rows)
      - Rest of discussion will focus on this case

When should tuples be constructed?

Solution 1: Create Rows first (EM)



QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

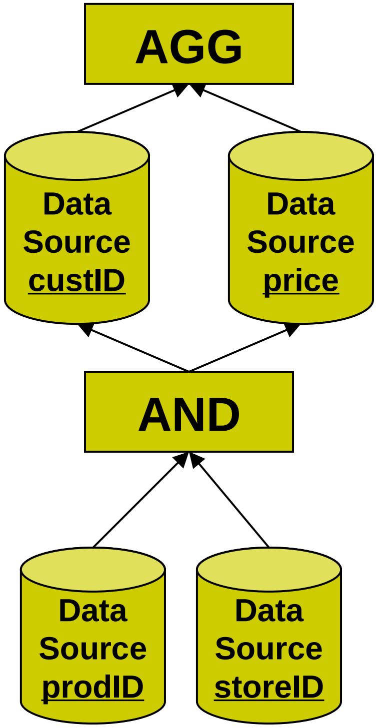
(storeID = 1)

GROUP BY custID

- **But:**
  - **Need to construct ALL tuples**
  - **Need to decompress data**
  - **Poor memory bandwidth utilization**

When should tuples be constructed?

Solution 2: Operate on columns



QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

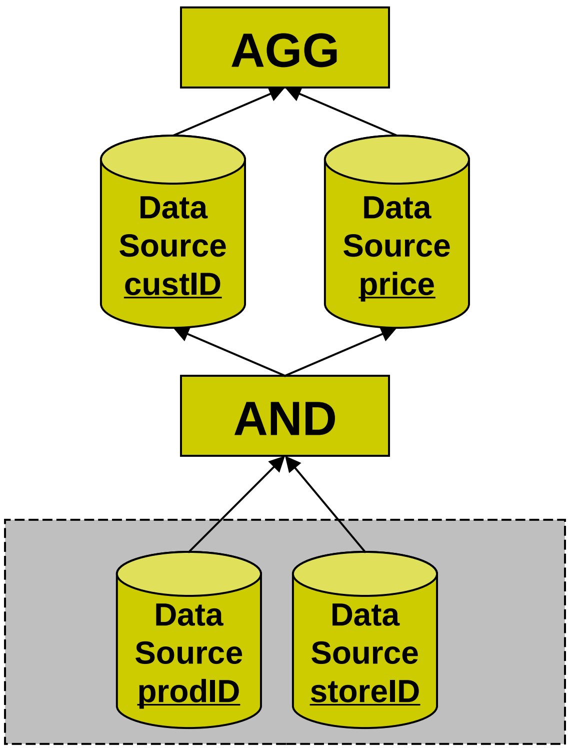
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

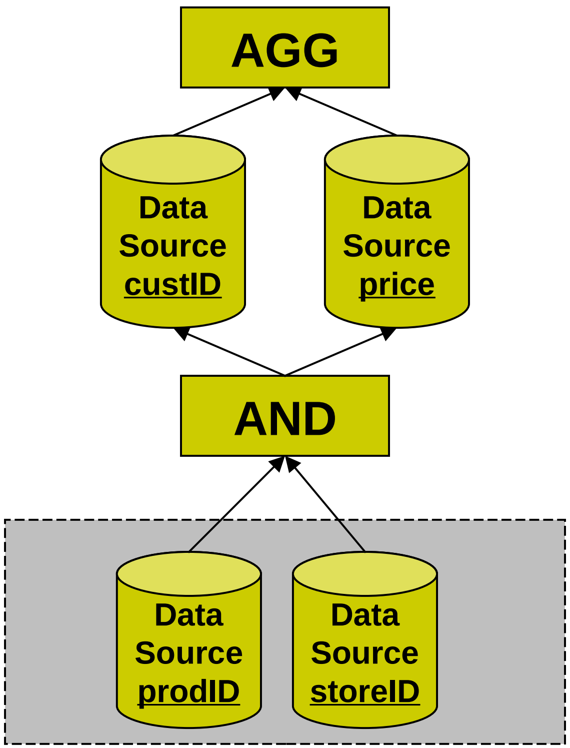
GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns

```
QUERY:
SELECT custID,SUM(price)
FROM table
WHERE (prodID = 4) AND
      (storeID = 1)
GROUP BY custID
```

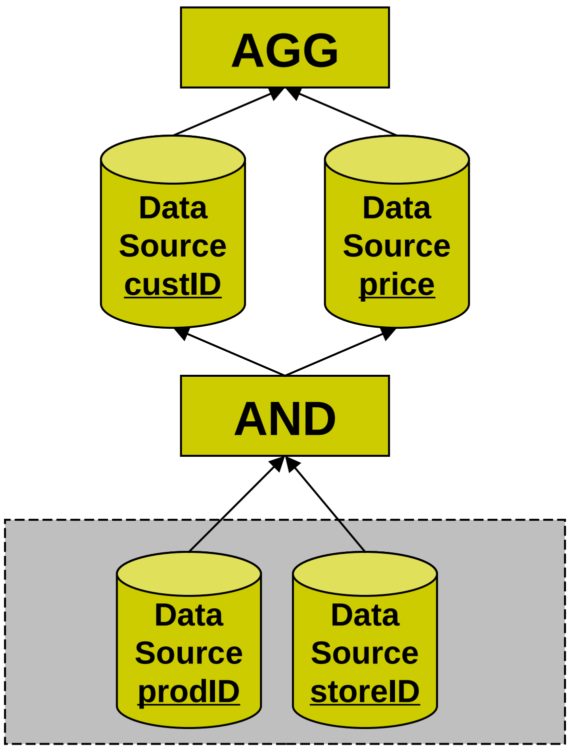


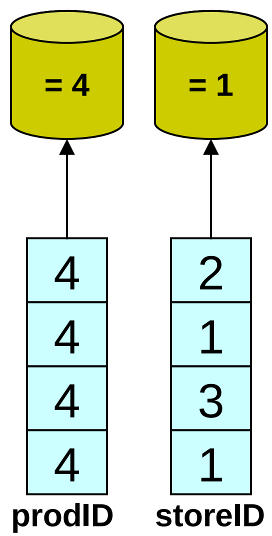
prodID	storeID
4	2
4	1
4	3
4	1

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns





QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

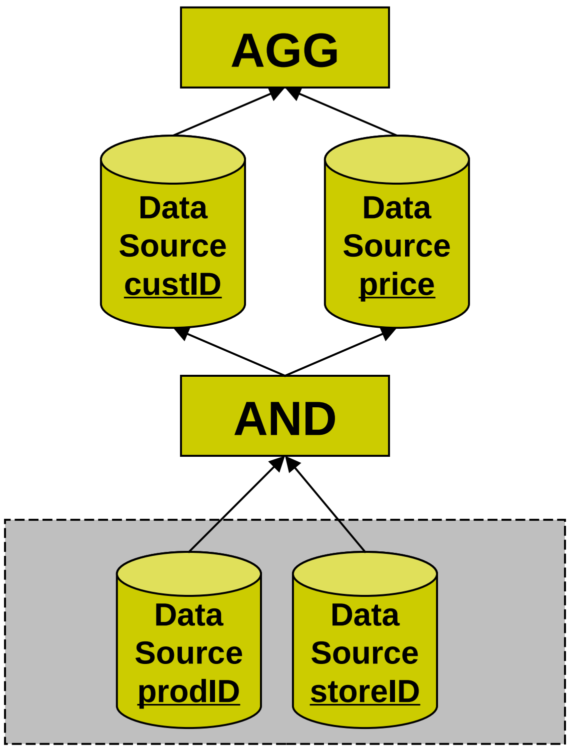
(storeID = 1)

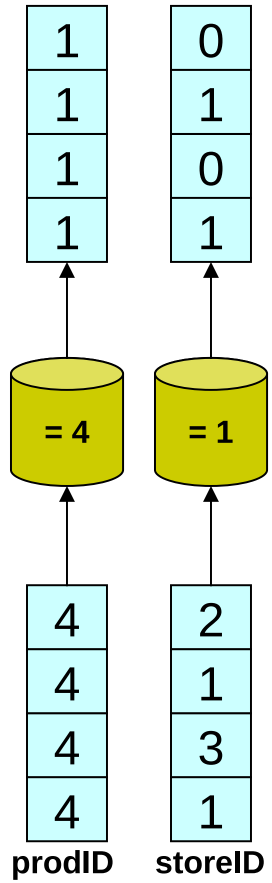
GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns





QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

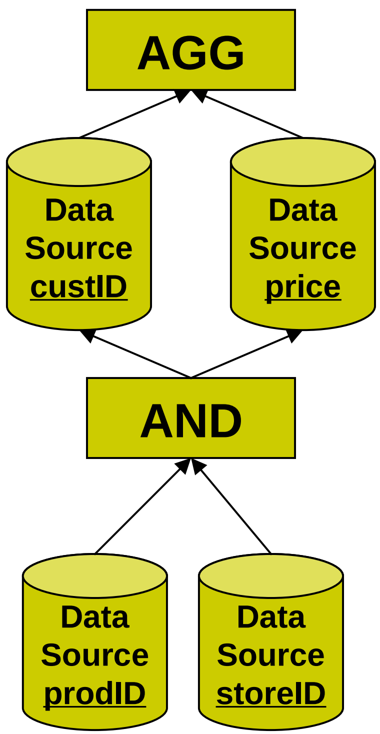
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



1	0
1	1
1	0
1	1

QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns

QUERY:

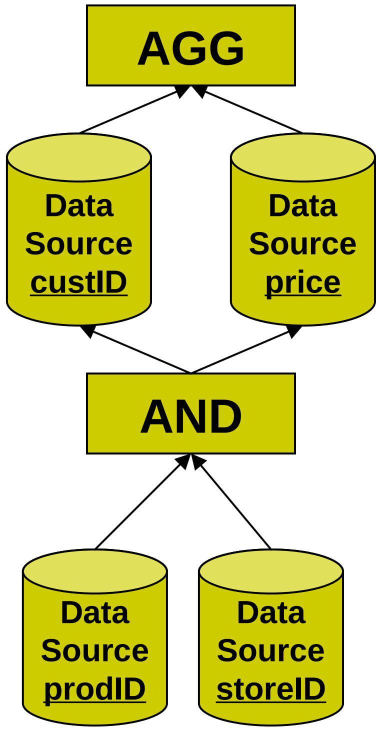
SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID

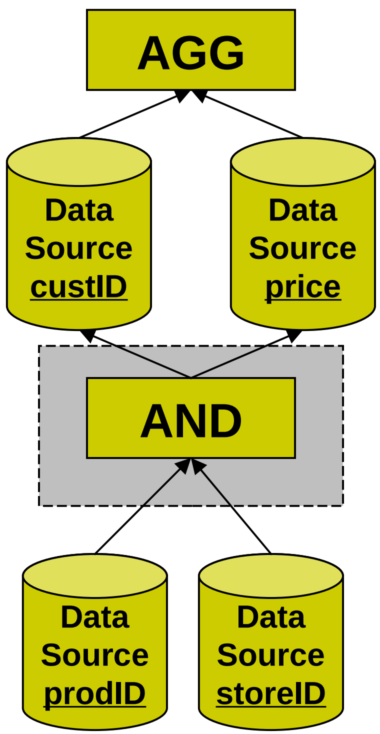


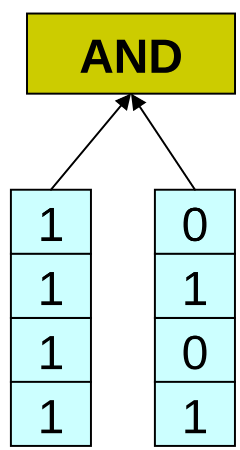
1	0
1	1
1	0
1	1

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns





QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

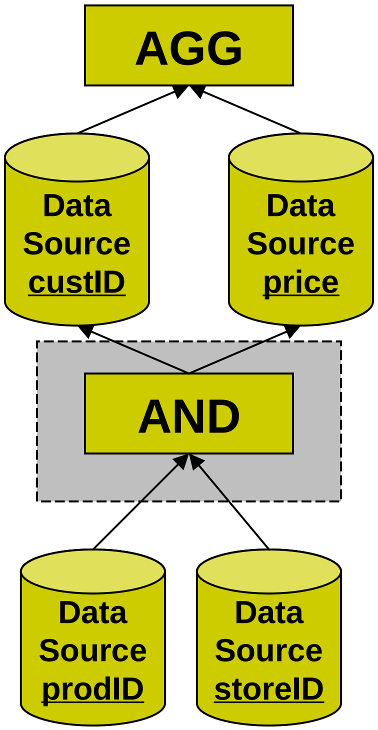
(storeID = 1)

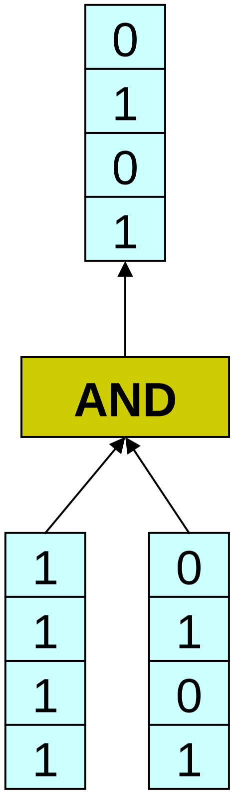
GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns





QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

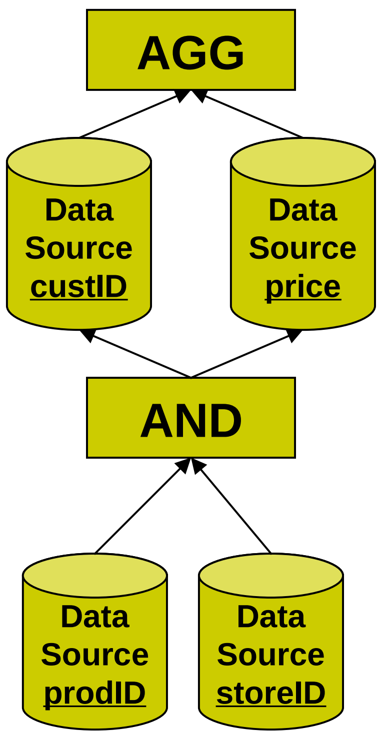
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



0
1
0
1

QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

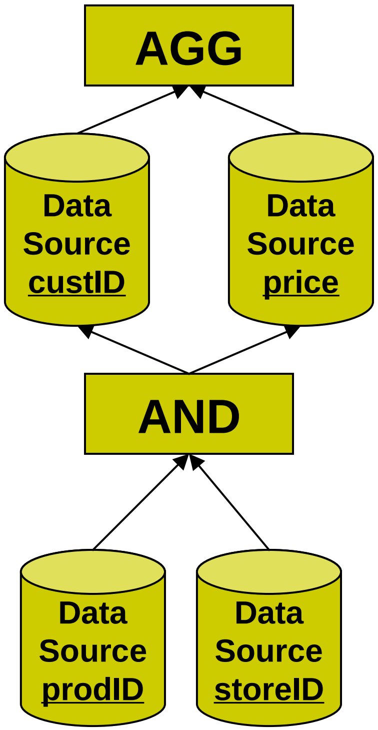
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



0
1
0
1

QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

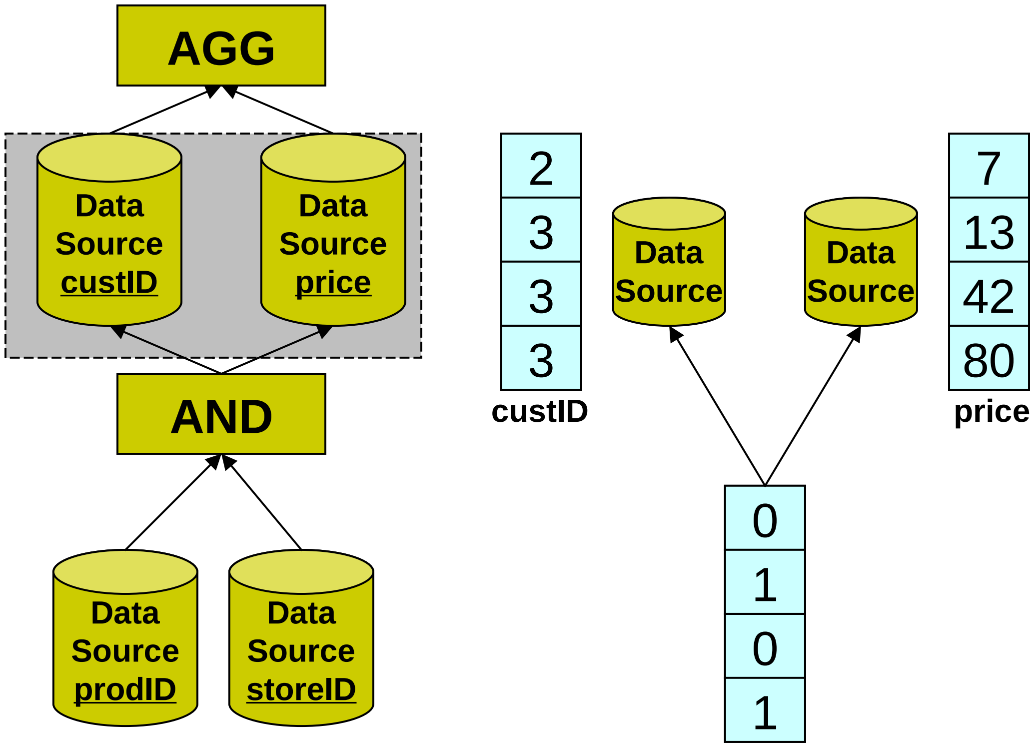
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



QUERY:

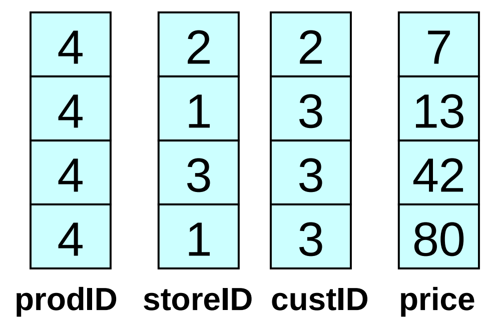
SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

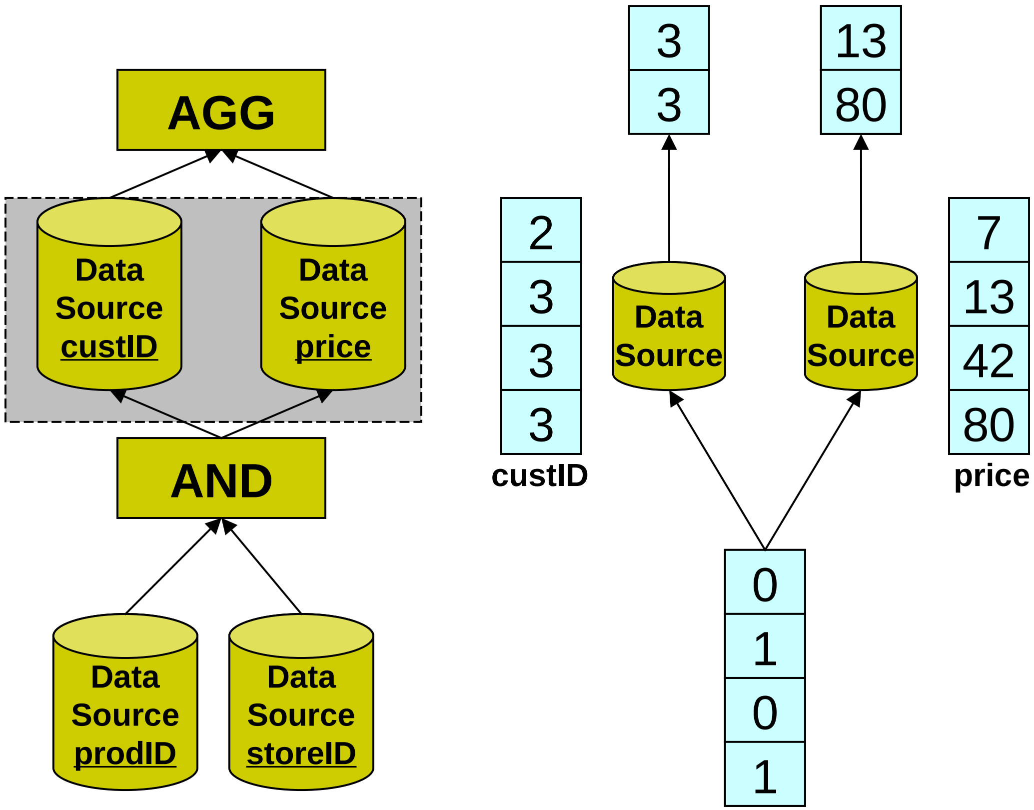
(storeID = 1)

GROUP BY custID



When should tuples be constructed?

Solution 2: Operate on columns



QUERY:

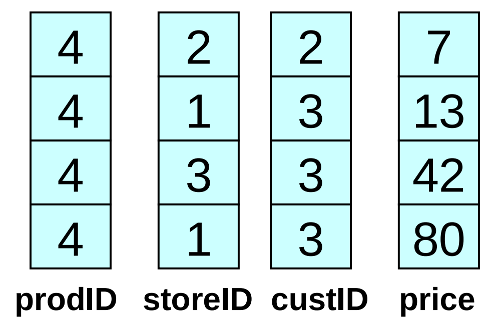
SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

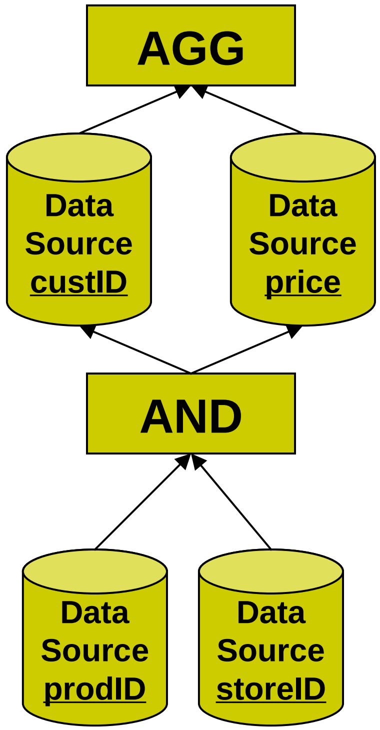
(storeID = 1)

GROUP BY custID



When should tuples be constructed?

Solution 2: Operate on columns



3	13
3	80

QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns

QUERY:

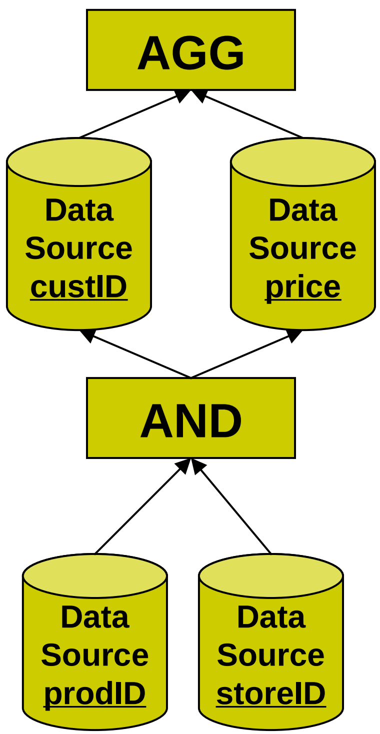
SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID



3
3

13
80

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns

QUERY:

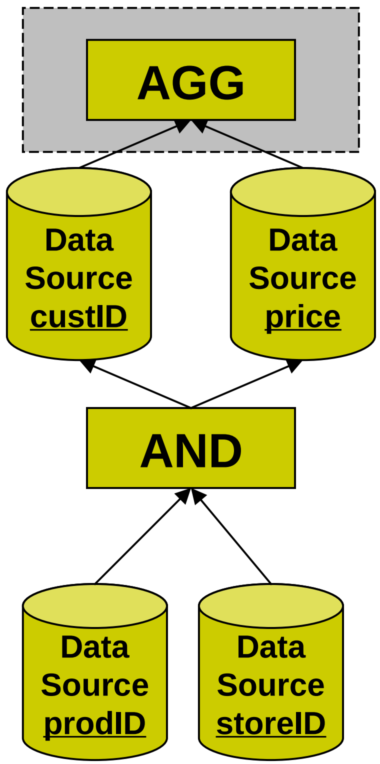
SELECT custID,SUM(price)

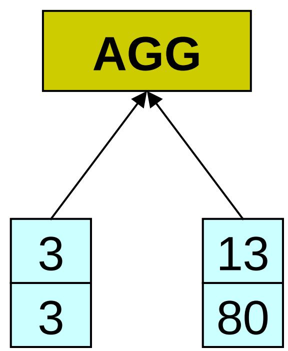
FROM table

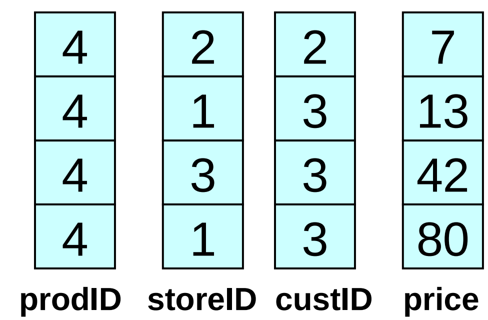
WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID

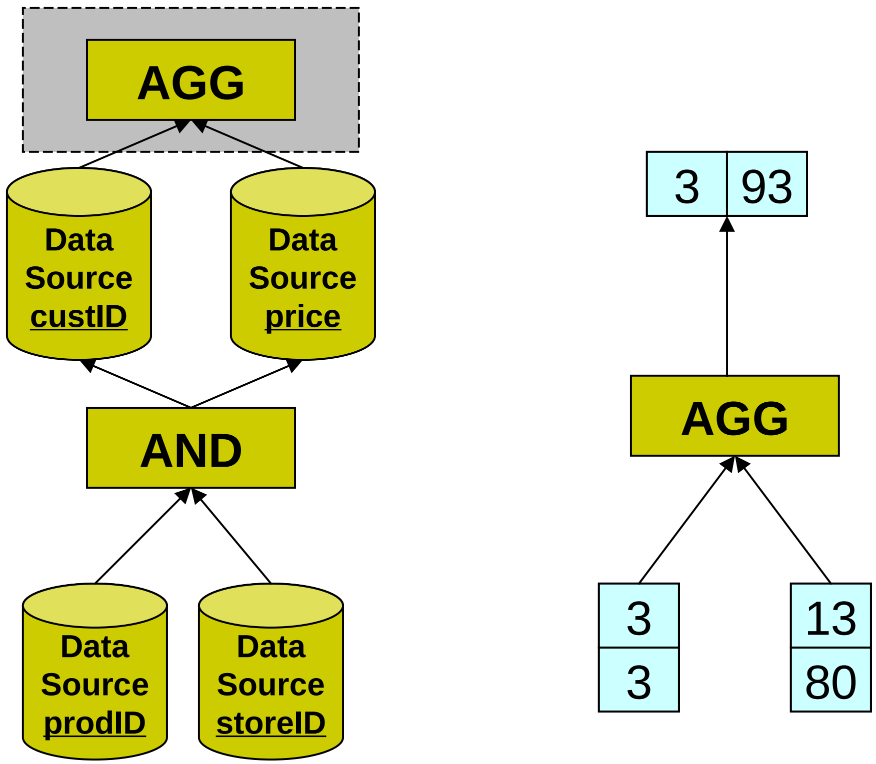






When should tuples be constructed?

Solution 2: Operate on columns



QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

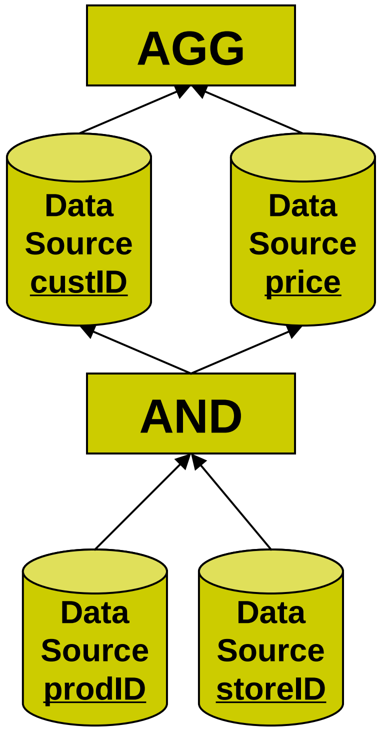
(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

When should tuples be constructed?

Solution 2: Operate on columns



3	93

QUERY:

SELECT custID,SUM(price)

FROM table

WHERE (prodID = 4) AND

(storeID = 1)

GROUP BY custID

prodID	storeID	custID	price
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

RISC Relational Algebra (MonetDB)

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

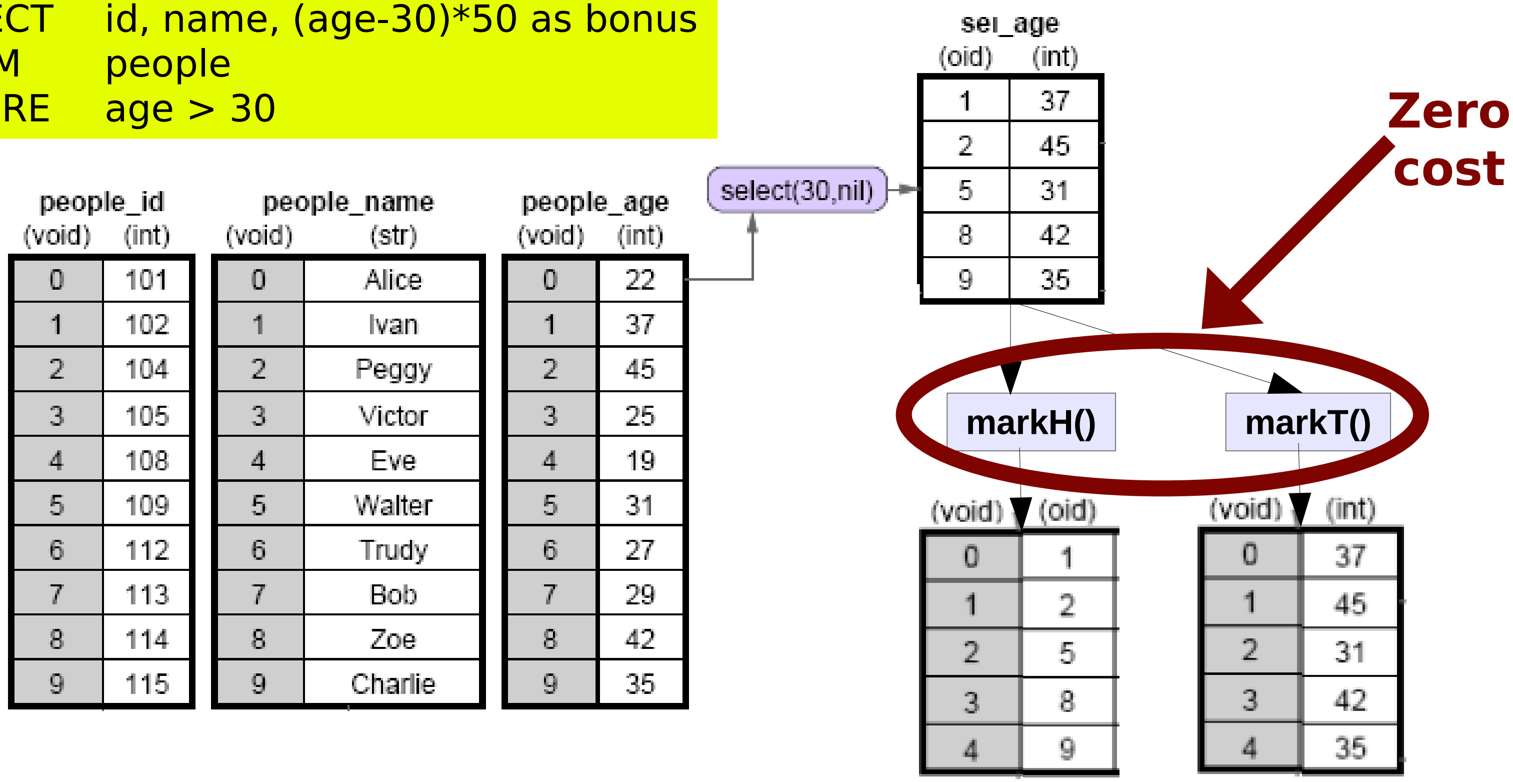
(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

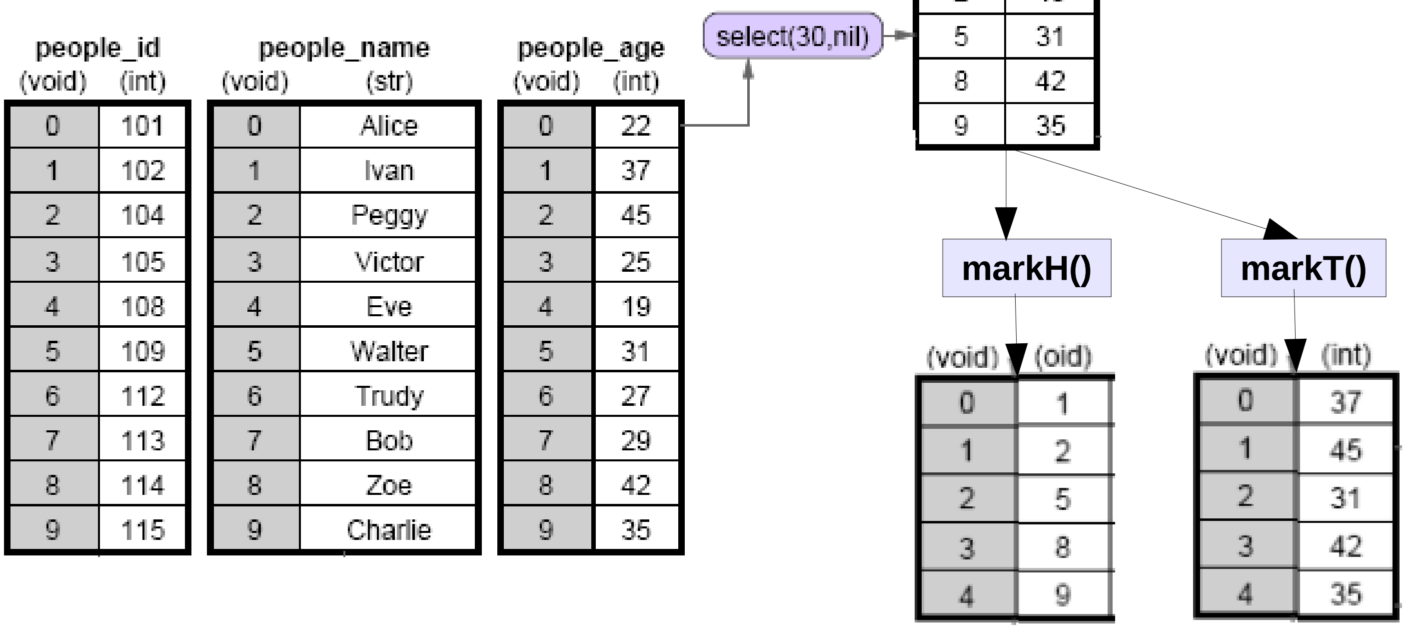
RISC Relational Algebra (MonetDB)

```
SELECT    id, name, (age-30)*50 as bonus
FROM      people
WHERE     age > 30
```



RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

VIEWS
(not materialized)

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

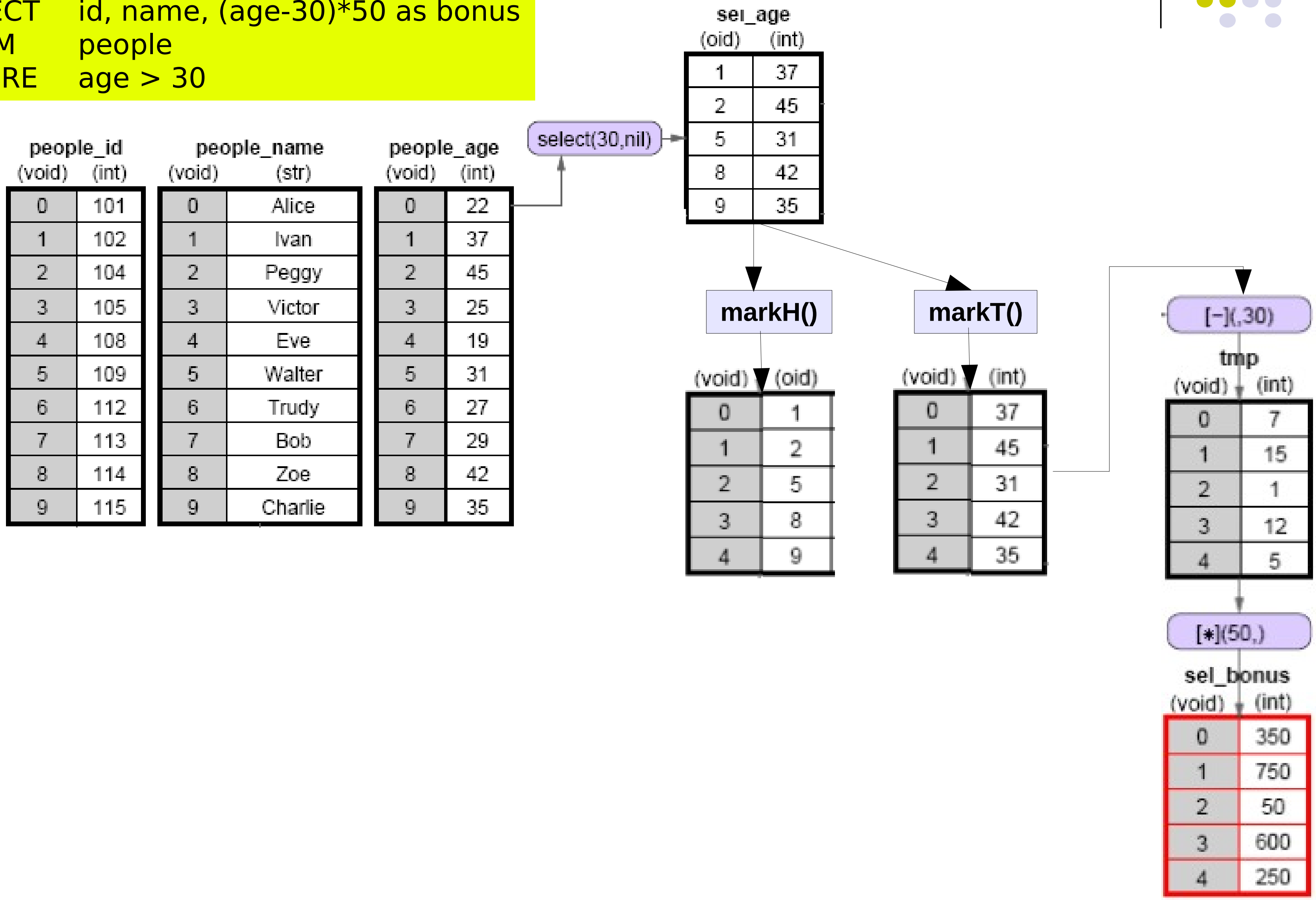
(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

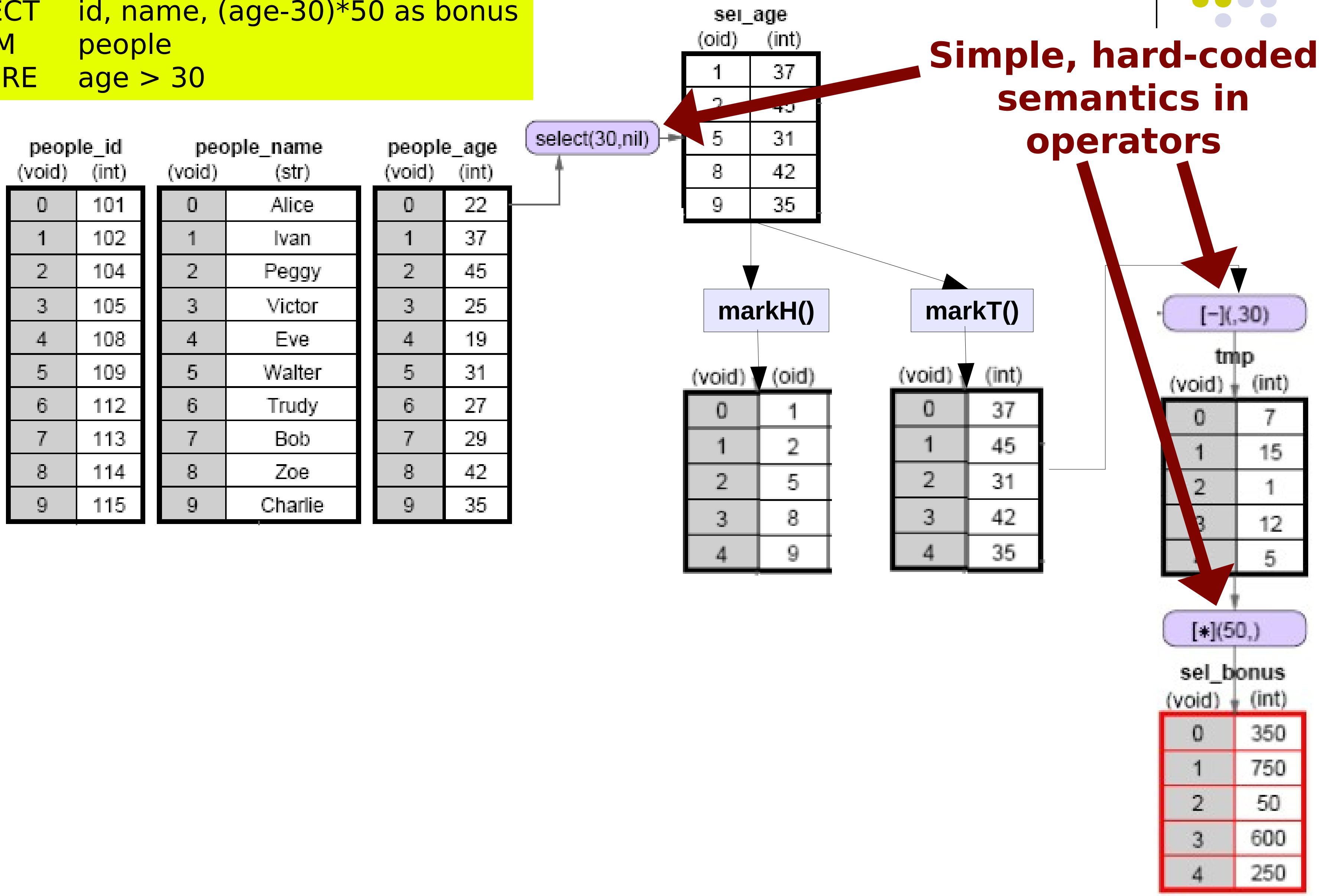
RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



RISC Relational Algebra (MonetDB)

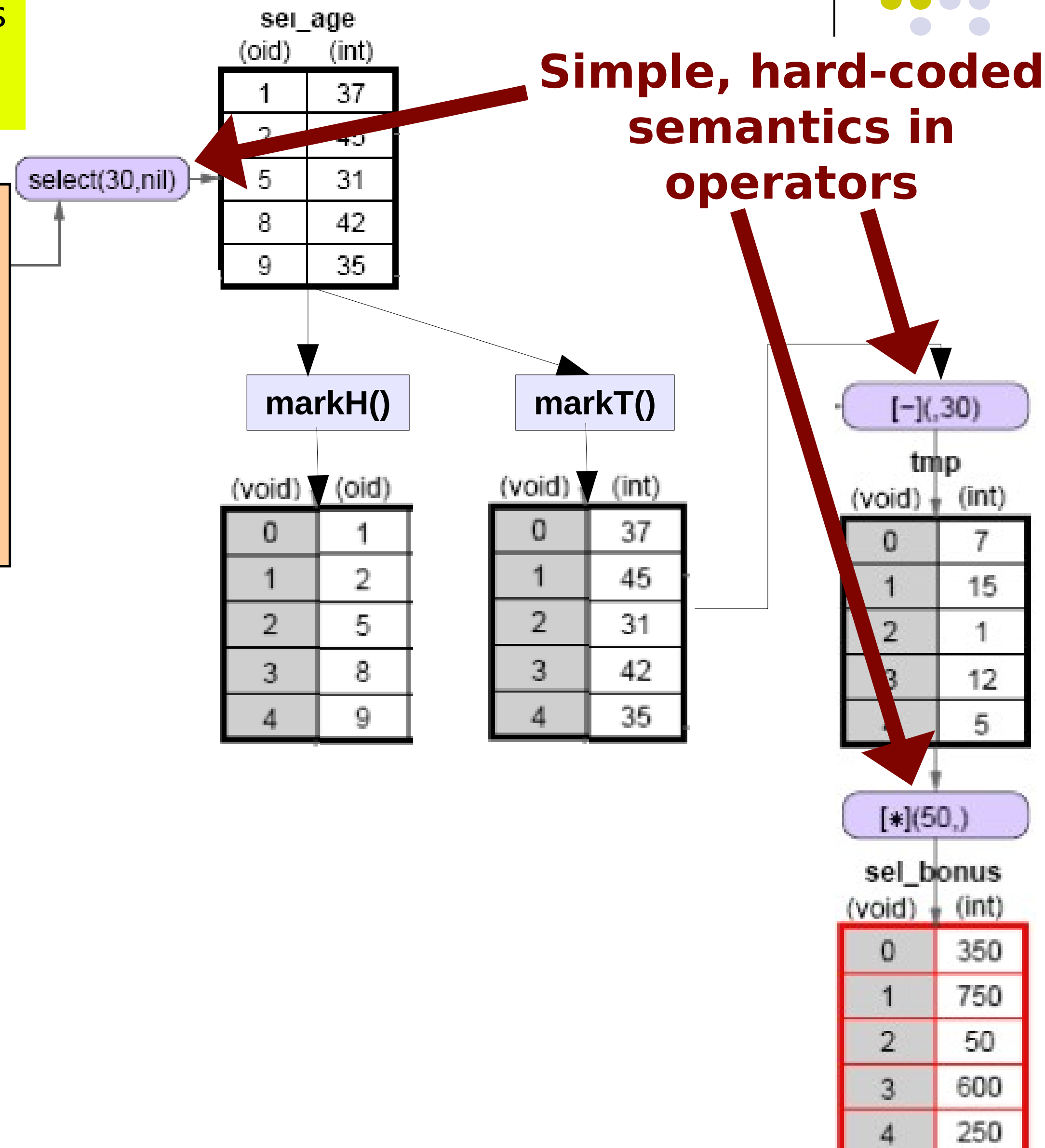
```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



RISC Relational Algebra (MonetDB)

```sql
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

```c
batcalc_minus_int(int* res,
                  int* col,
                  int  val,
                  int n)
{
    for(i=0; i<n; i++)
        res[i] = col[i] - val;
}
```



Simple, hard-coded semantics in operators

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

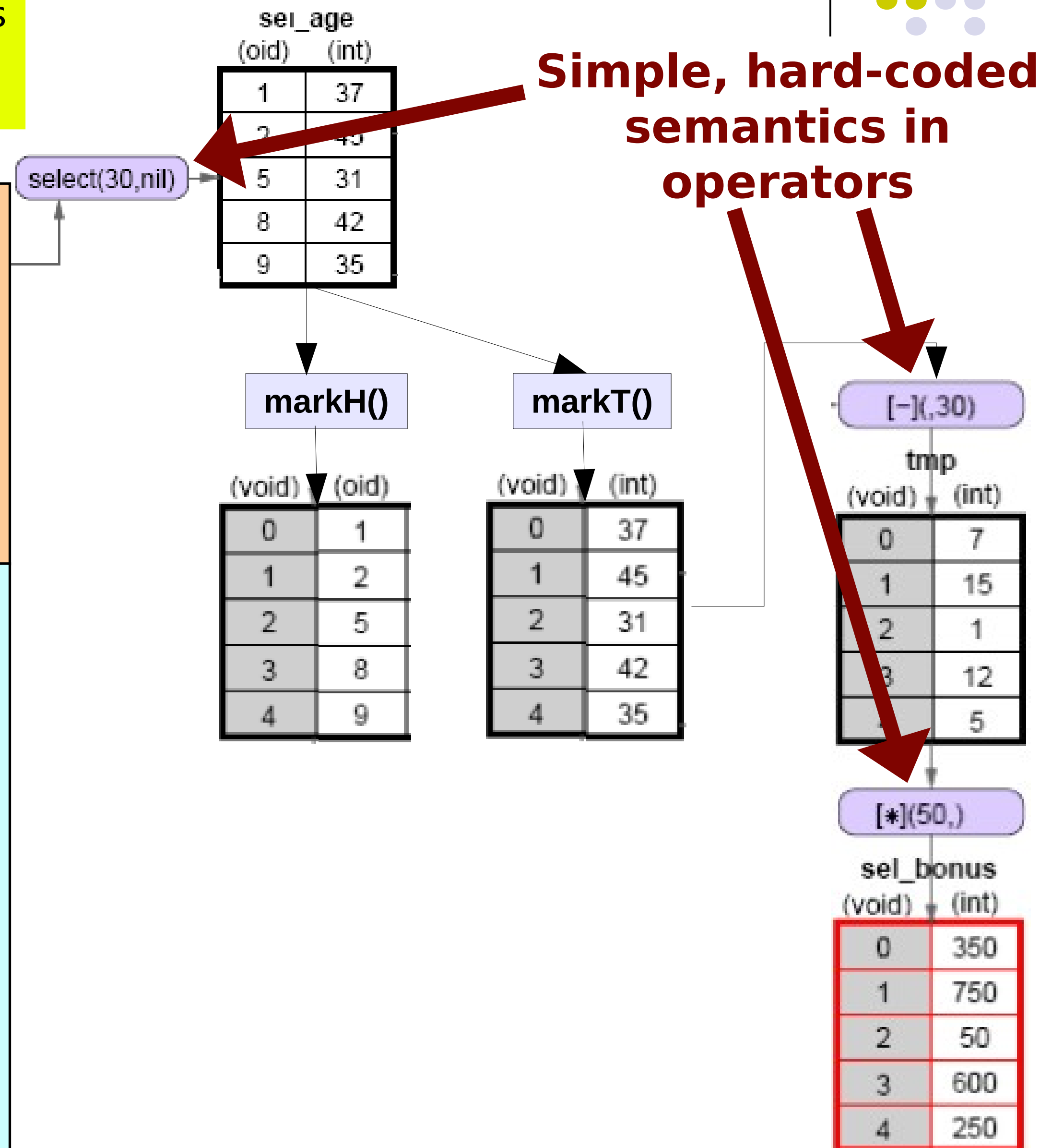
```
batcalc_minus_int(int* res,
                  int* col,
                  int  val,
                  int n)
{
    for(i=0; i<n; i++)
        res[i] = col[i] - val;
}
```

CPU ☺? Give it "nice" code !

- few dependencies (control,data)
- CPU gets out-of-order execution
- compiler can e.g. generate SIMD

One loop for an entire column

- no per-tuple interpretation
- arrays: no record navigation
- better instruction cache locality



RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

[−](,30)

tmp

(void)	(int)
0	7
1	15
2	1
3	12
4	5

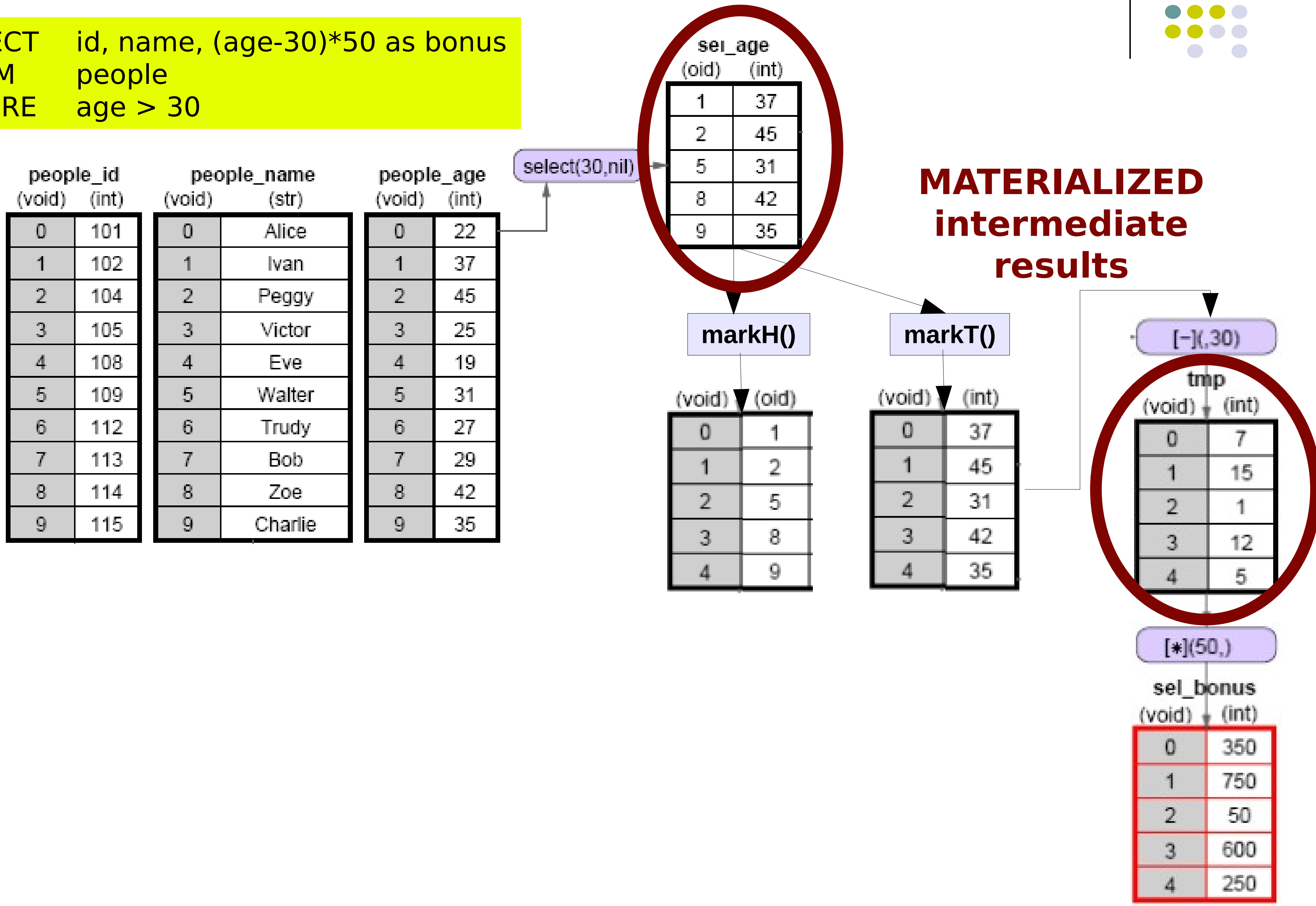
[∗](50,)

sel_bonus

(void)	(int)
0	350
1	750
2	50
3	600
4	250

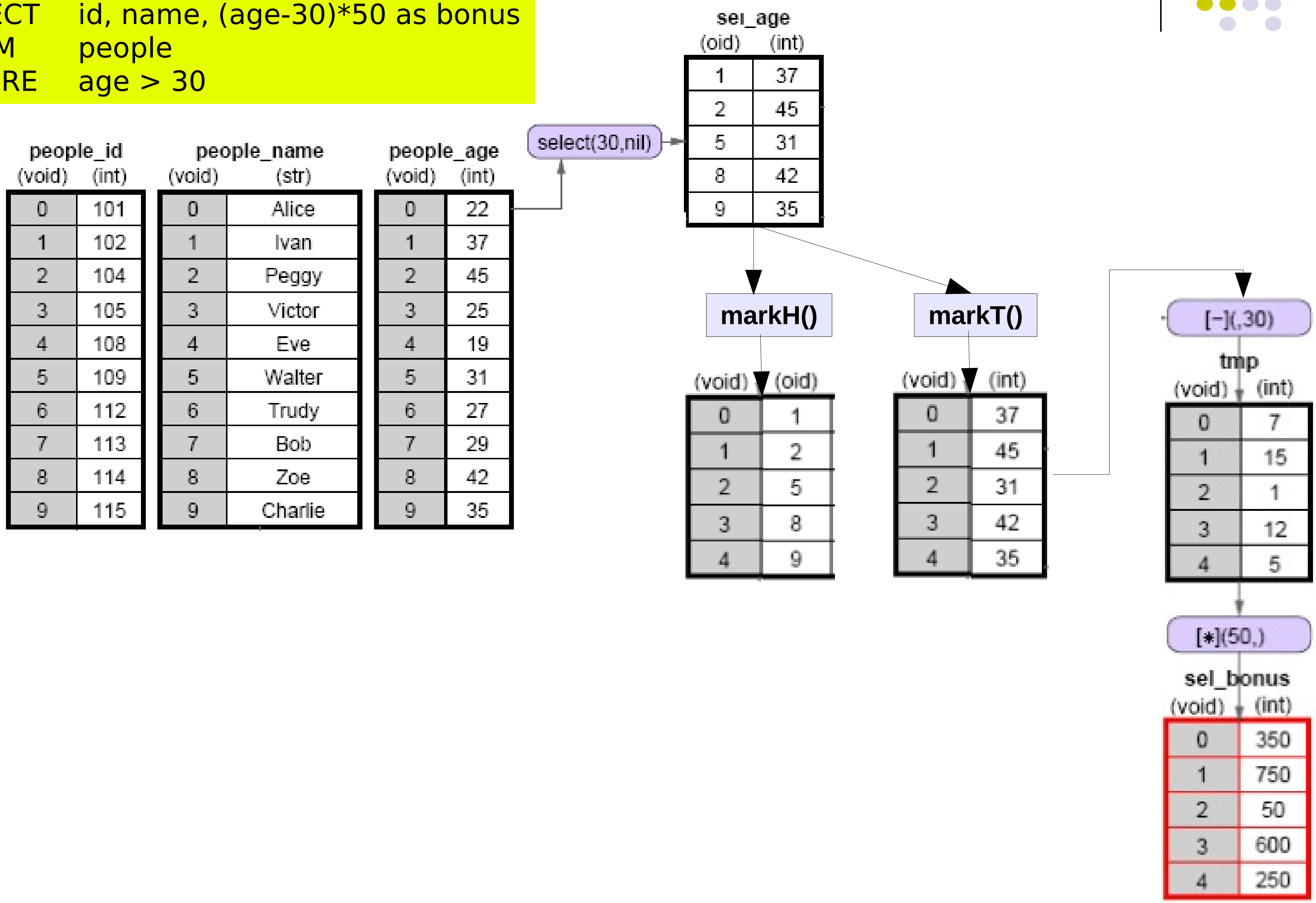
RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



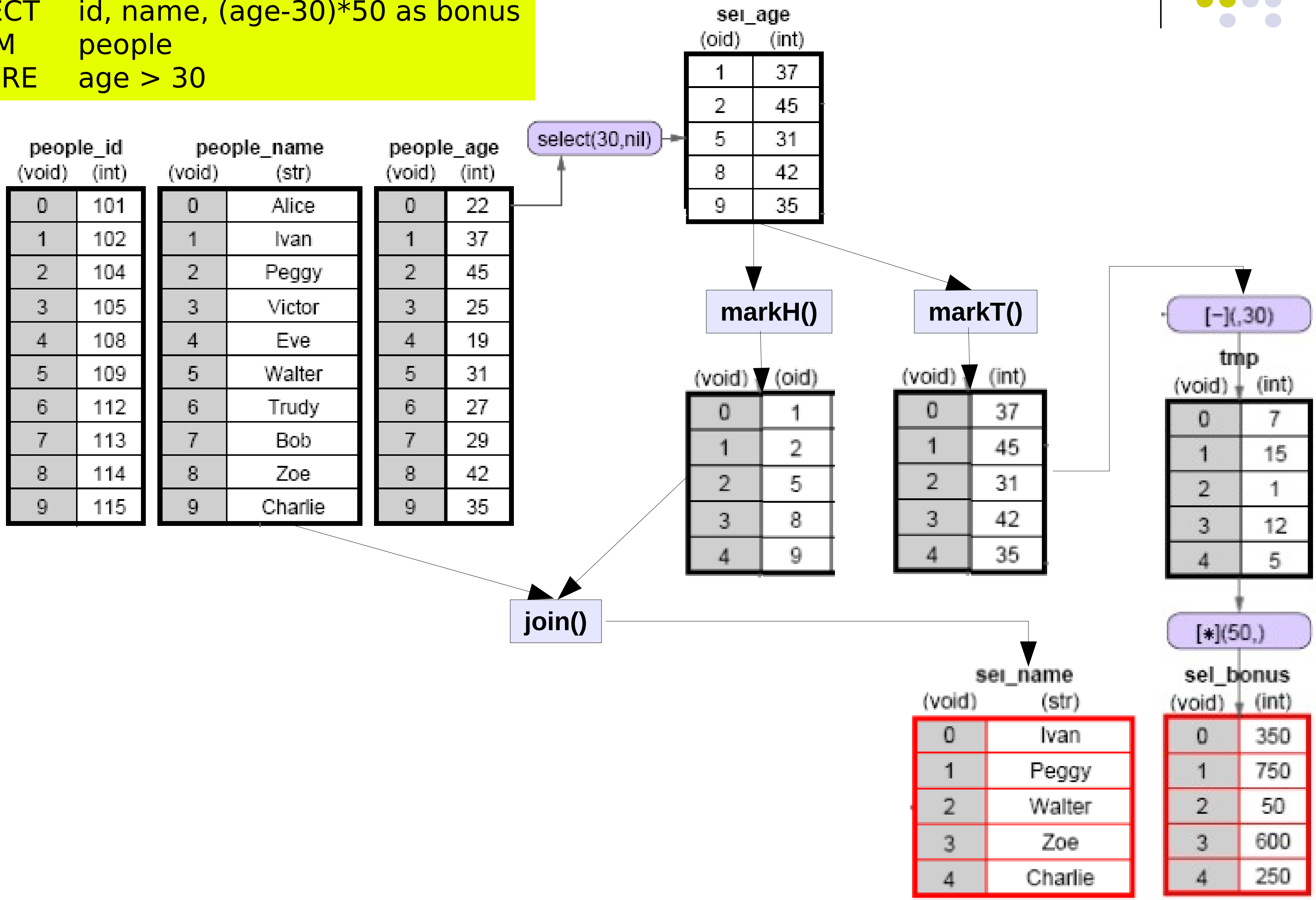
RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



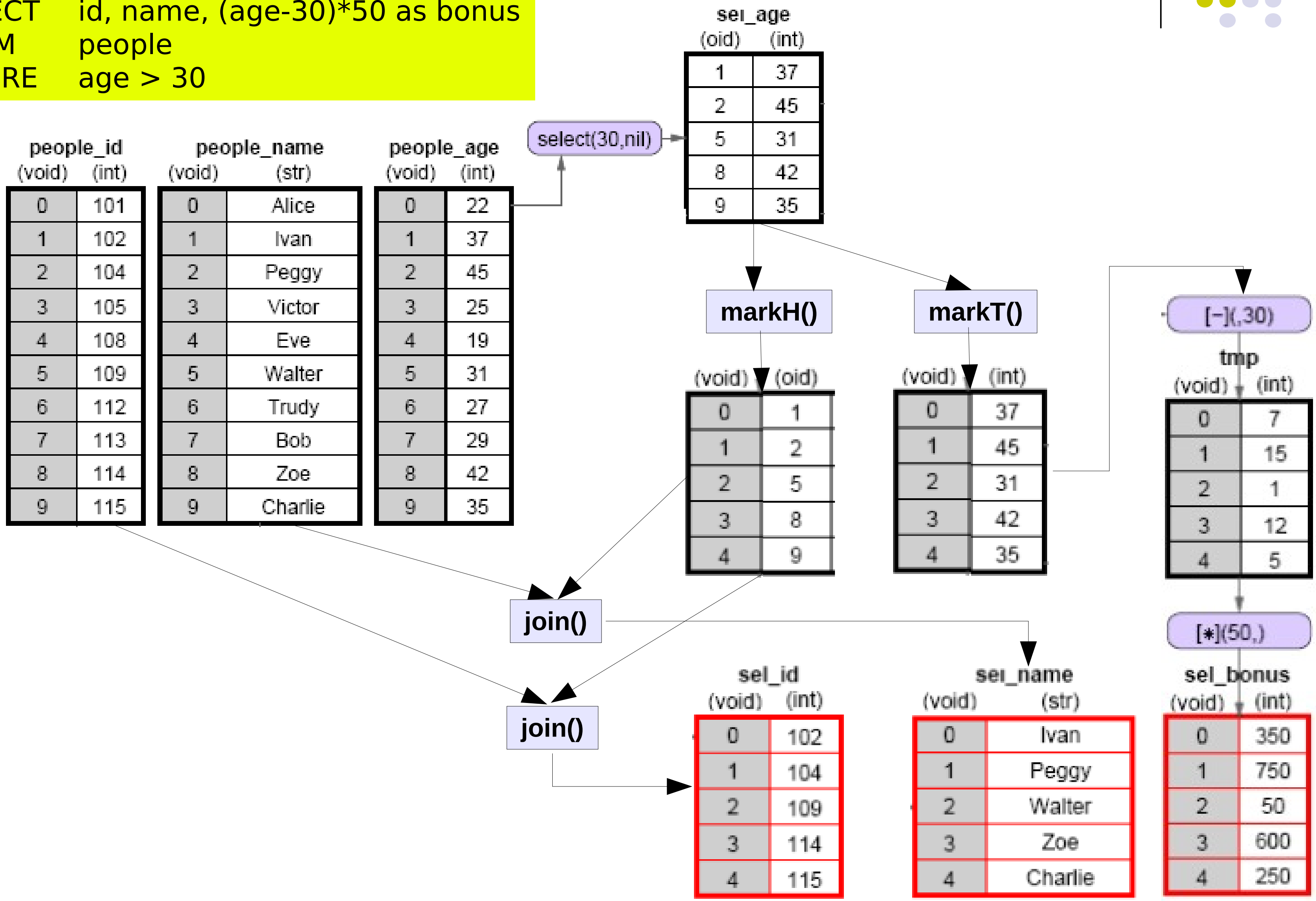
RISC Relational Algebra (MonetDB)

SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30



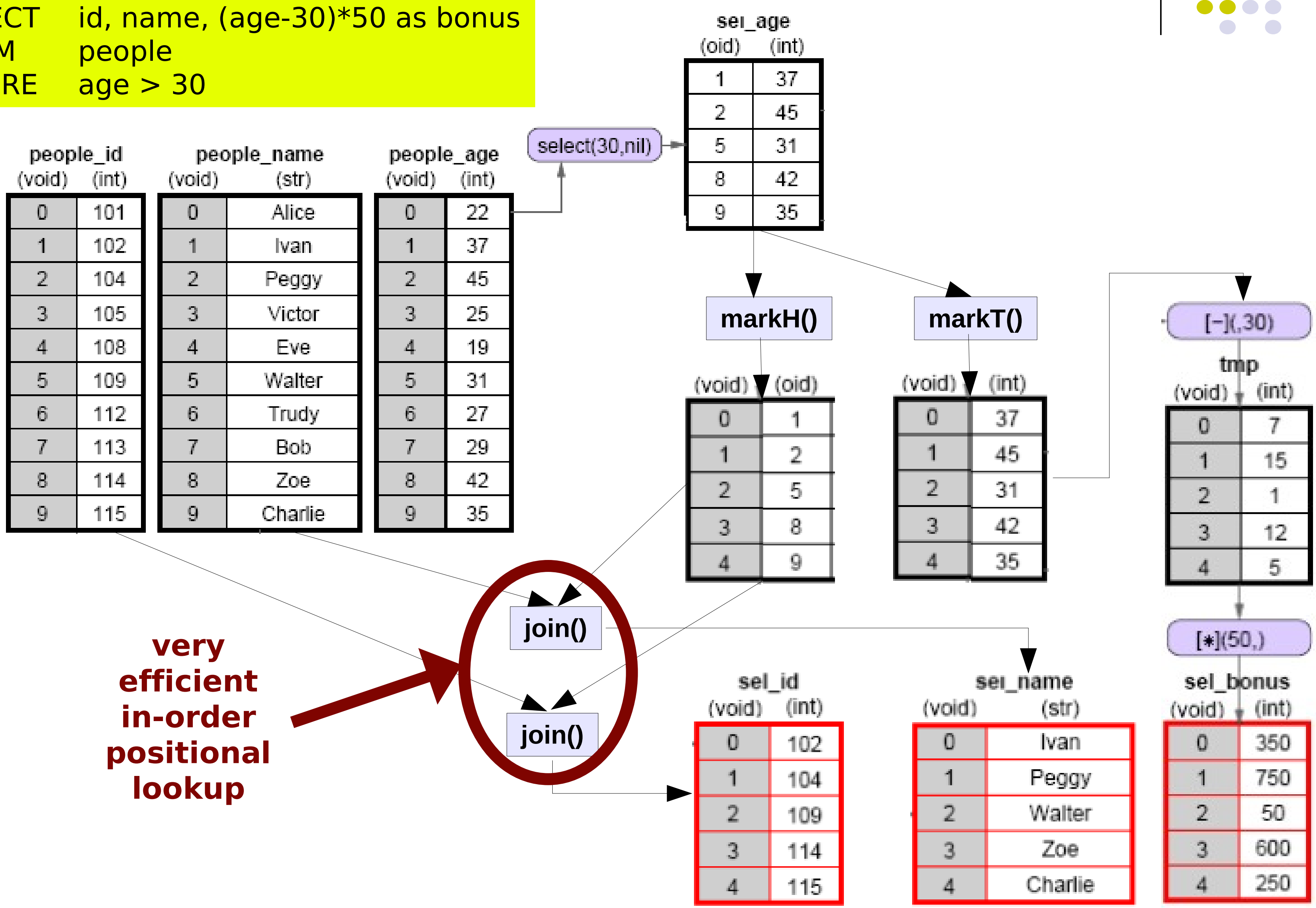
RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

tmp

(void)	(int)
0	7
1	15
2	1
3	12
4	5

sel_id

(void)	(int)
0	102
1	104
2	109
3	114
4	115

sel_name

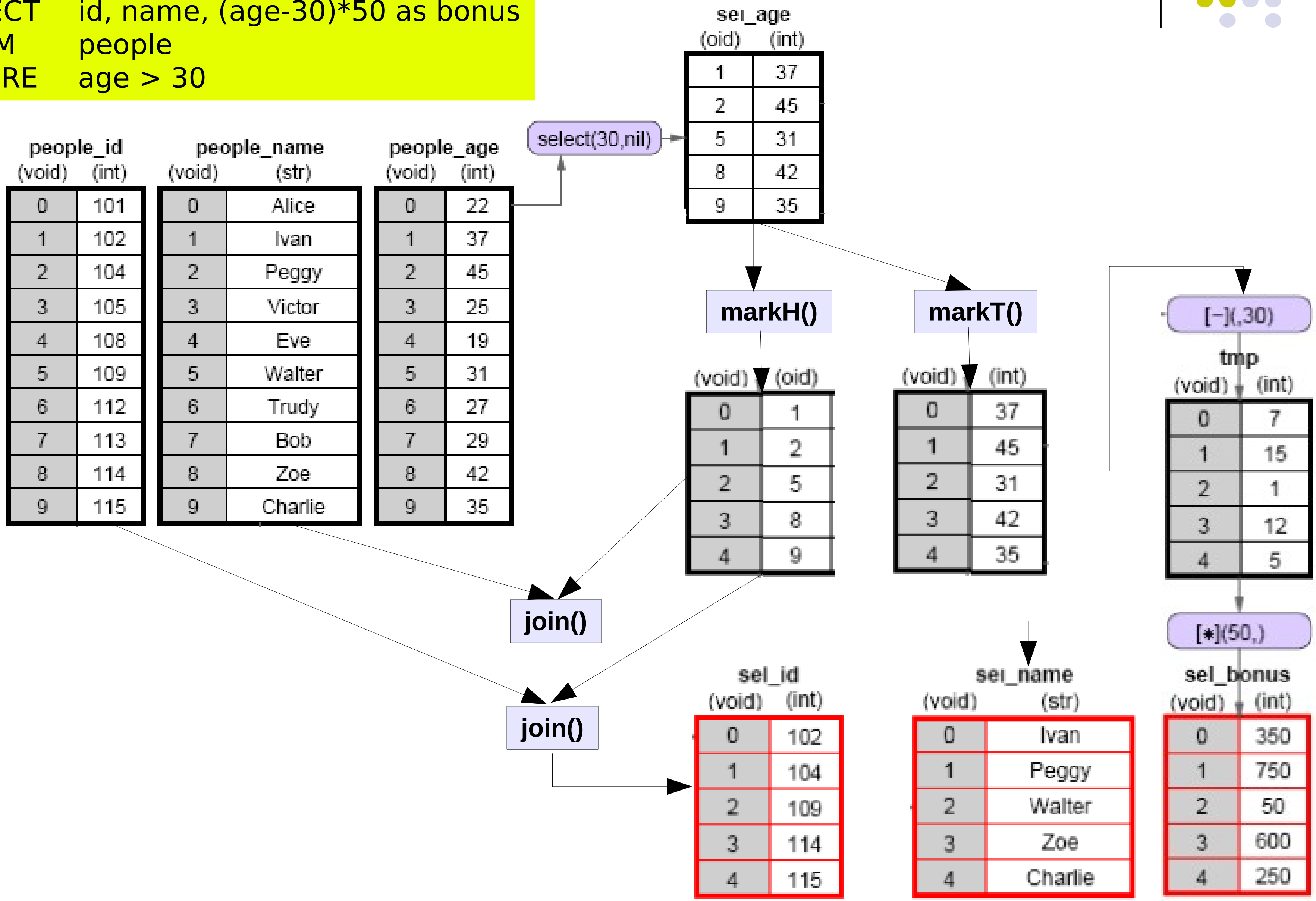
(void)	(str)
0	Ivan
1	Peggy
2	Walter
3	Zoe
4	Charlie

sel_bonus

(void)	(int)
0	350
1	750
2	50
3	600
4	250

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```



RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

[-](,30)

tmp

(void)	(int)
0	7
1	15
2	1
3	12
4	5

[*](50,)

```
_01 := sql.bind("sys","people","age",0);
_02 := algebra.select(_01,30,nil);
_03 := algebra.markH(_02,0);
_04 := algebra.markT(_02,0);
_05 := batcalc.-(_04,30);
_06 := batcalc.*(_05,50);
_07 := sql.bind("sys","people","name",0);
_08 := algebra.join(_03,_07);
_09 := sql.bind("sys","people","id",0);
_10 := algebra.join(_03,_09);
```

sel_id

(void)	(int)
0	102
1	104
2	109
3	114
4	115

sel_name

(void)	(str)
0	Ivan
1	Peggy
2	Walter
3	Zoe
4	Charlie

sel_bonus

(void)	(int)
0	350
1	750
2	50
3	600
4	250

RISC Relational Algebra (MonetDB)

```
SELECT   id, name, (age-30)*50 as bonus
FROM     people
WHERE    age > 30
```

people_id

(void)	(int)
0	101
1	102
2	104
3	105
4	108
5	109
6	112
7	113
8	114
9	115

people_name

(void)	(str)
0	Alice
1	Ivan
2	Peggy
3	Victor
4	Eve
5	Walter
6	Trudy
7	Bob
8	Zoe
9	Charlie

people_age

(void)	(int)
0	22
1	37
2	45
3	25
4	19
5	31
6	27
7	29
8	42
9	35

select(30,nil)

sel_age

(oid)	(int)
1	37
2	45
5	31
8	42
9	35

markH()

(void)	(oid)
0	1
1	2
2	5
3	8
4	9

markT()

(void)	(int)
0	37
1	45
2	31
3	42
4	35

[-](,30)

tmp

(void)	(int)
0	7
1	15
2	1
3	12
4	5

[*](50,)

sel_id

(void)	(int)
0	102
1	104
2	109
3	114
4	115

sel_name

(void)	(str)
0	Ivan
1	Peggy
2	Walter
3	Zoe
4	Charlie

sel_bonus

(void)	(int)
0	350
1	750
2	50
3	600
4	250

```
_01 := sql.bind("sys","people","age",0);
_02 := algebra.select(_01,30,nil);
_03 := algebra.markH(_02,0);
_04 := algebra.markT(_02,0);
_05 := batcalc.-(_04,30);
_06 := batcalc.*(_05,50);
_07 := sql.bind("sys","people","name",0);
_08 := algebra.project(_03,_07);
_09 := sql.bind("sys","people","id",0);
_10 := algebra.project(_03,_09);
```

MonetDB Front-end: SQL

```
PLAN SELECT a FROM t WHERE c < 10;

project (
  select (
    table(sys.t) [ t.a, t.c, t.%TID% NOT NULL ]
  ) [ t.c < convert(10) ]
) [ t.a ]
```

MonetDB Front-end: SQL

EXPLAIN SELECT a FROM t WHERE c < 10;

```
function user.s1_1():void;
barrier _55 := language.dataflow();
    _02:bat[:void,:int] := sql.bind("sys","t","c",0);
    _07:bat[:oid, :int] := algebra.thetauselect(_02,10,"<");
    _11:bat[:void,:oid] := algebra.markH(_07,0@0);
    _12:bat[:oid, :int] := sql.bind("sys","t","a",0);
    _14:bat[:void,:int] := algebra.project(_11,_12);
exit _55;
    _15 := sql.resultSet(1,1,_14);
    sql.rsColumn(_15,"sys.t","a","int",32,0,_14);
    _21 := io.stdout();
    sql.exportResult(_21,_15);
end s1_1;
```

MonetDB Front-end: SQL

```
PLAN SELECT a, z FROM t, s WHERE t.c = s.x;
PLAN SELECT a, z FROM t join s on t.c = s.x;

project (
  join (
    table(sys.t) [ t.a, t.c, t.%TID% NOT NULL ],
    table(sys.s) [ s.x, s.z, s.%TID% NOT NULL ]
  ) [ t.c = s.x ]
) [ t.a, s.z ]
```

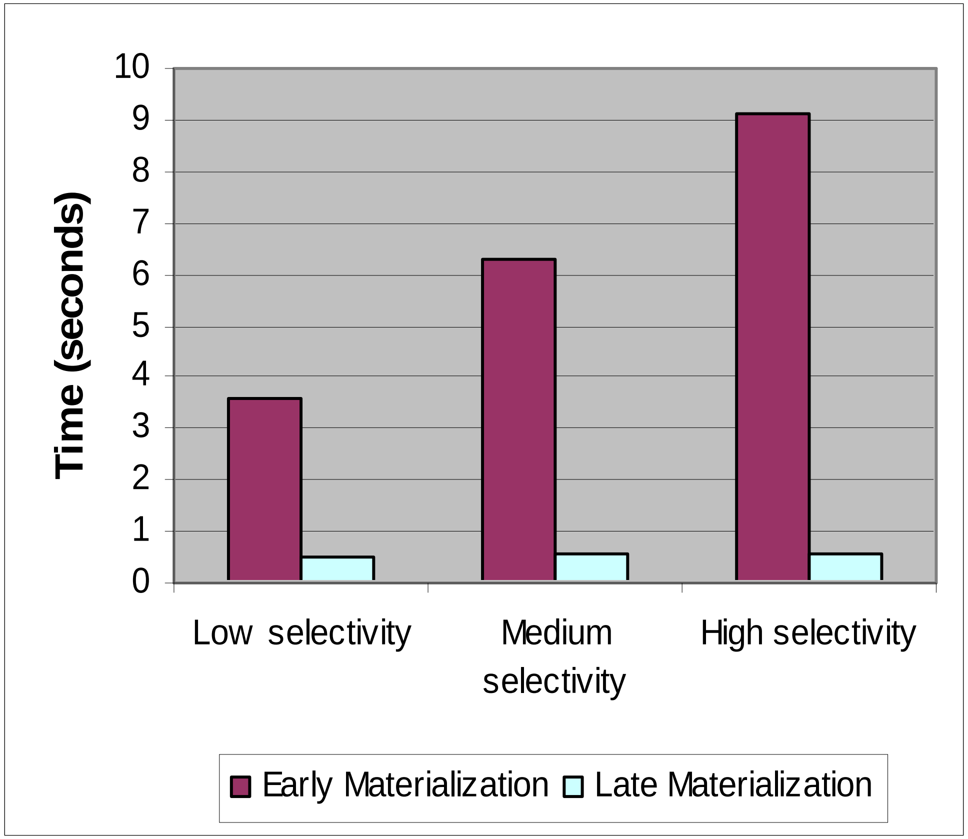
MonetDB Front-end: SQL

```
EXPLAIN SELECT a, z FROM t, s WHERE t.c = s.x;

function user.s2_1():void;
 barrier _73 := language.dataflow();
  _02:bat[:void,:int] := sql.bind("sys","t","c",0);
  _07:bat[:void,:int] := sql.bind("sys","s","x",0);
  _10:bat[:int,:void] := bat.reverse(_07);
  _11:bat[:oid, :oid] := algebra.join(_02,_10);
  _14:bat[:void,:oid] := algebra.markH(_11,0@0);
  _15:bat[:void,:int] := sql.bind("sys","t","a",0);
  _17:bat[:void,:int] := algebra.project(_14,_15);
  _20:bat[:oid,:void] := algebra.markT(_18,0@0);
  _21:bat[:void,:int] := sql.bind("sys","s","z",0);
  _23:bat[:void,:int] := algebra.project(_20,_21);
 exit _73;
  _24 := sql.resultSet(2,1,_17);
  sql.rsColumn(_24,"sys.t","a","int",32,0,_17);
  sql.rsColumn(_24,"sys.s","z","int",32,0,_23);
  _33 := io.stdout();
  sql.exportResult(_33,_24);
end s2_1;
```

"Materialization Strategies in a Column-Oriented DBMS"
Abadi, Myers, DeWitt, and Madden. ICDE 2007.

For plans without joins, late materialization is a win



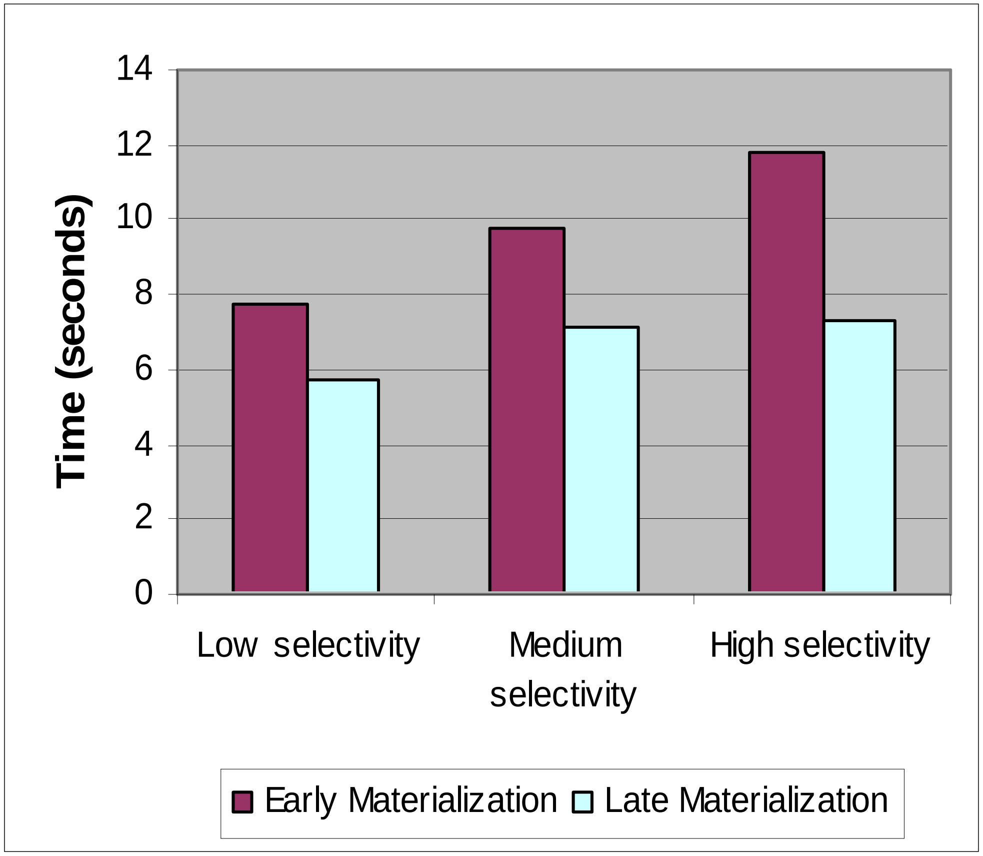
QUERY:

```
SELECT C1, SUM(C2)
FROM table
WHERE (C1 < CONST) AND
      (C2 < CONST)
GROUP BY C1
```

- **Ran on 2 compressed columns from TPC-H scale 10 data**

"Materialization Strategies in a Column-Oriented DBMS"
Abadi, Myers, DeWitt, and Madden. ICDE 2007.

Even on uncompressed data, late materialization is still a win



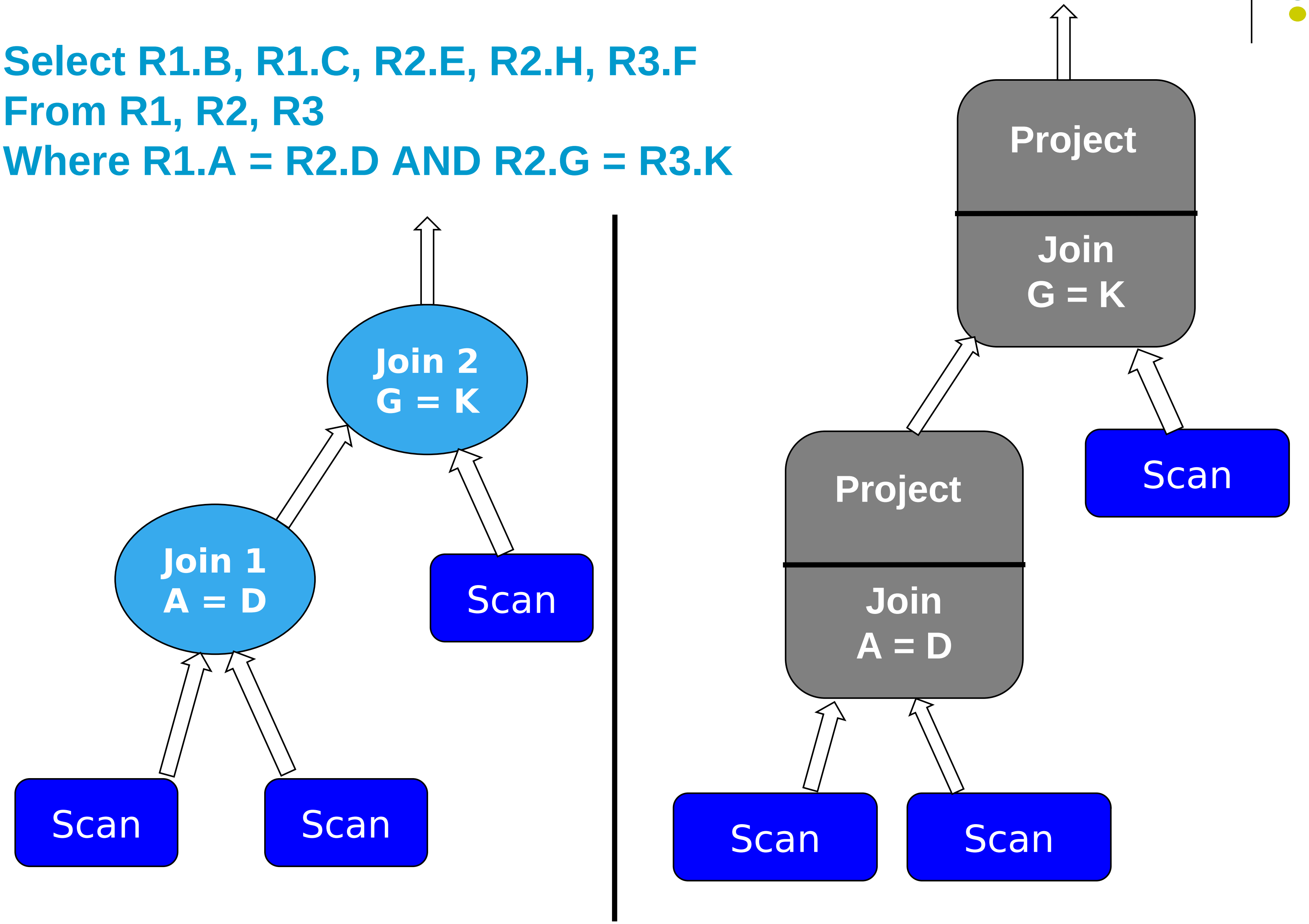
QUERY:

```
SELECT C1, SUM(C2)
FROM table
WHERE (C1 < CONST) AND
      (C2 < CONST)
GROUP BY C1
```

- **Materializing late still works best**

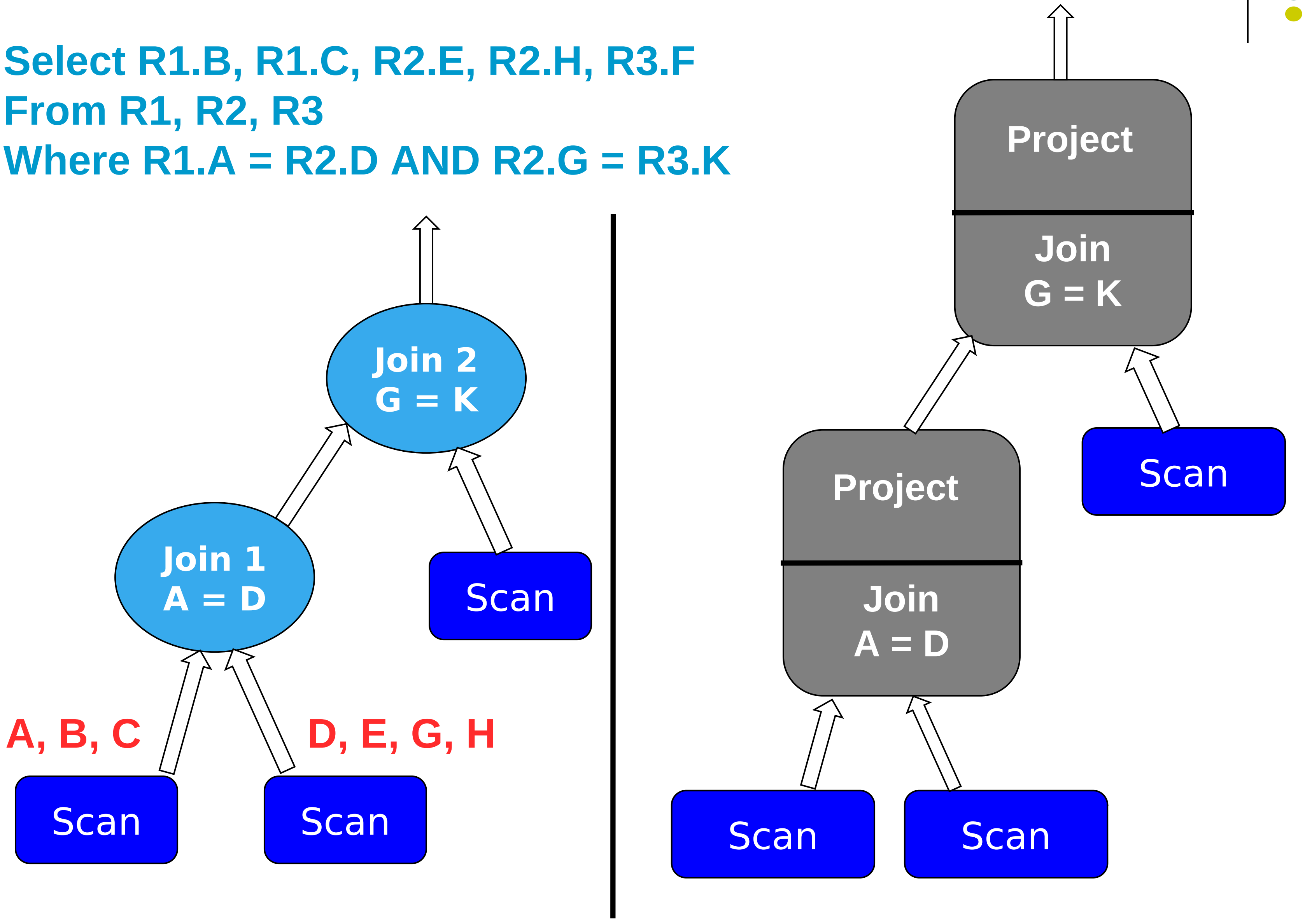
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



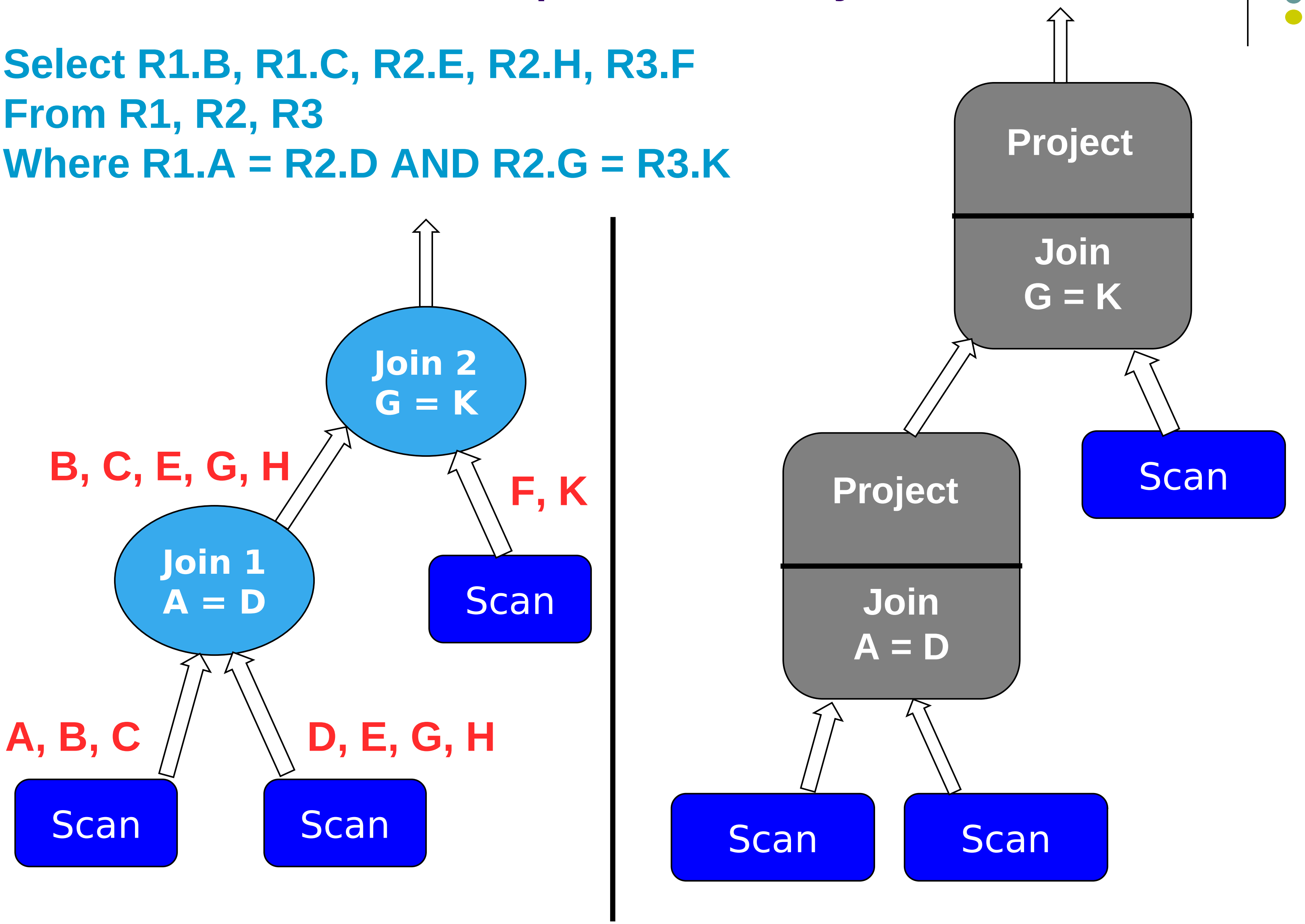
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



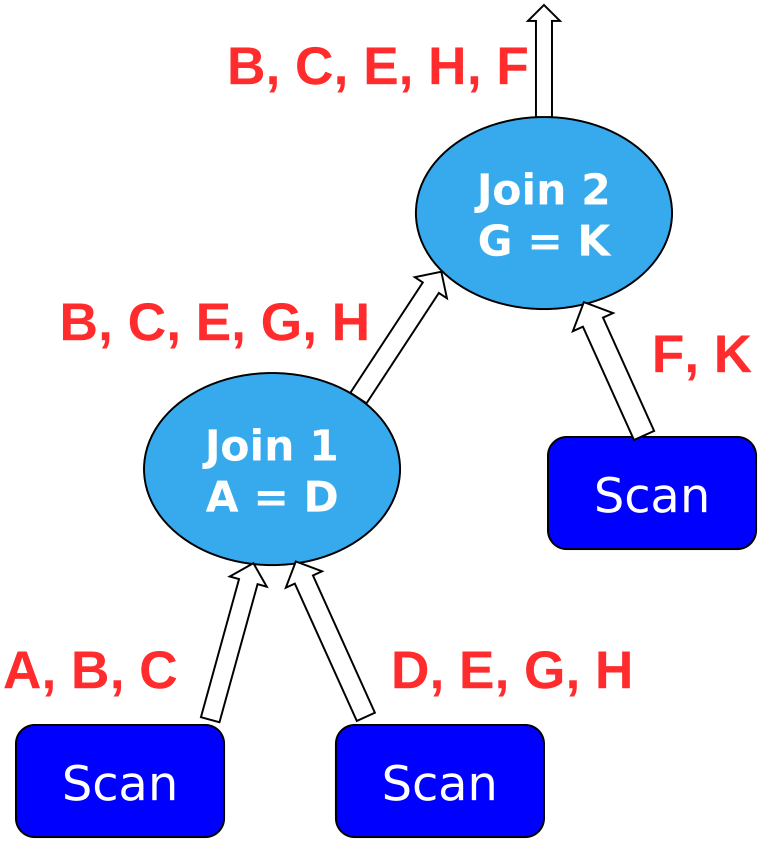
What about for plans with joins?

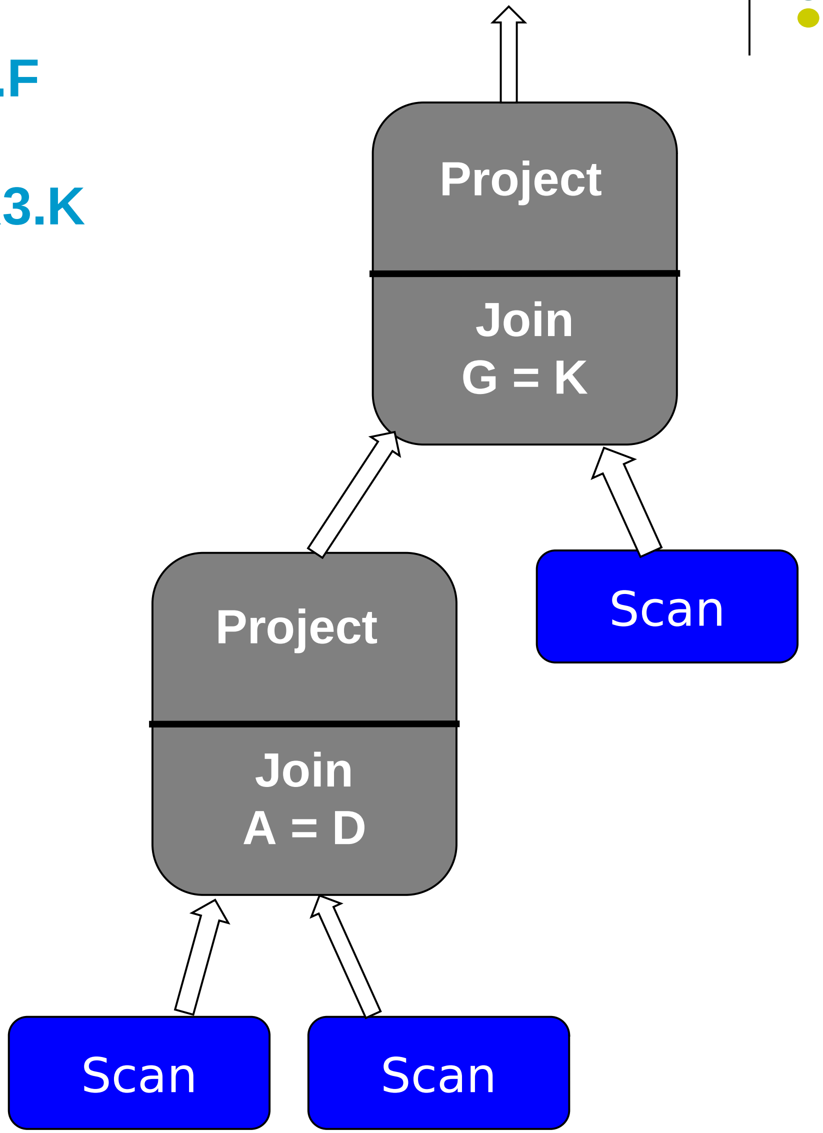
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



What about for plans with joins?

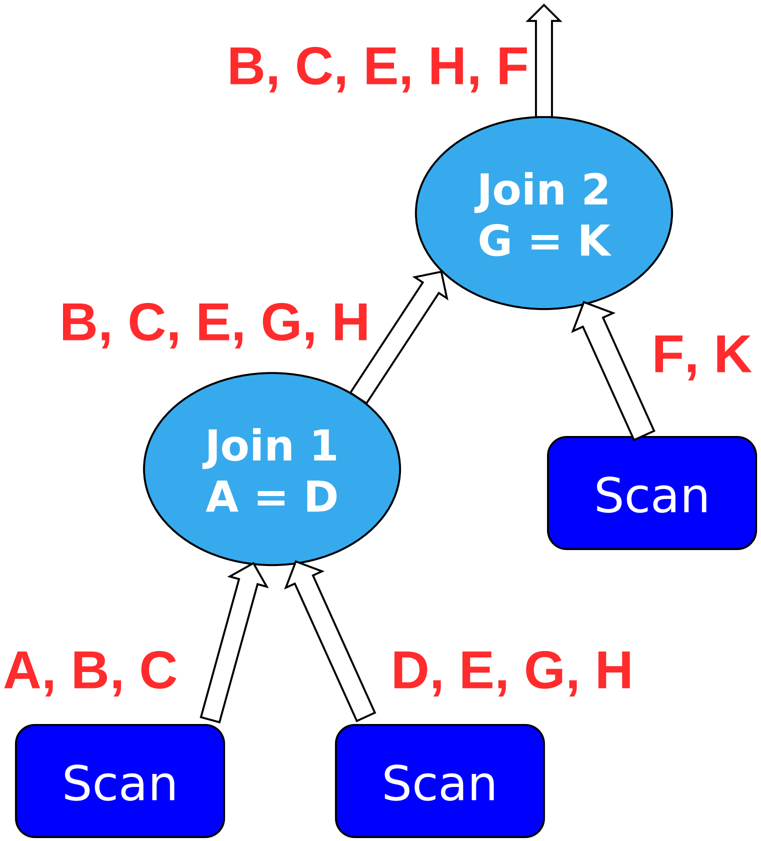
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K

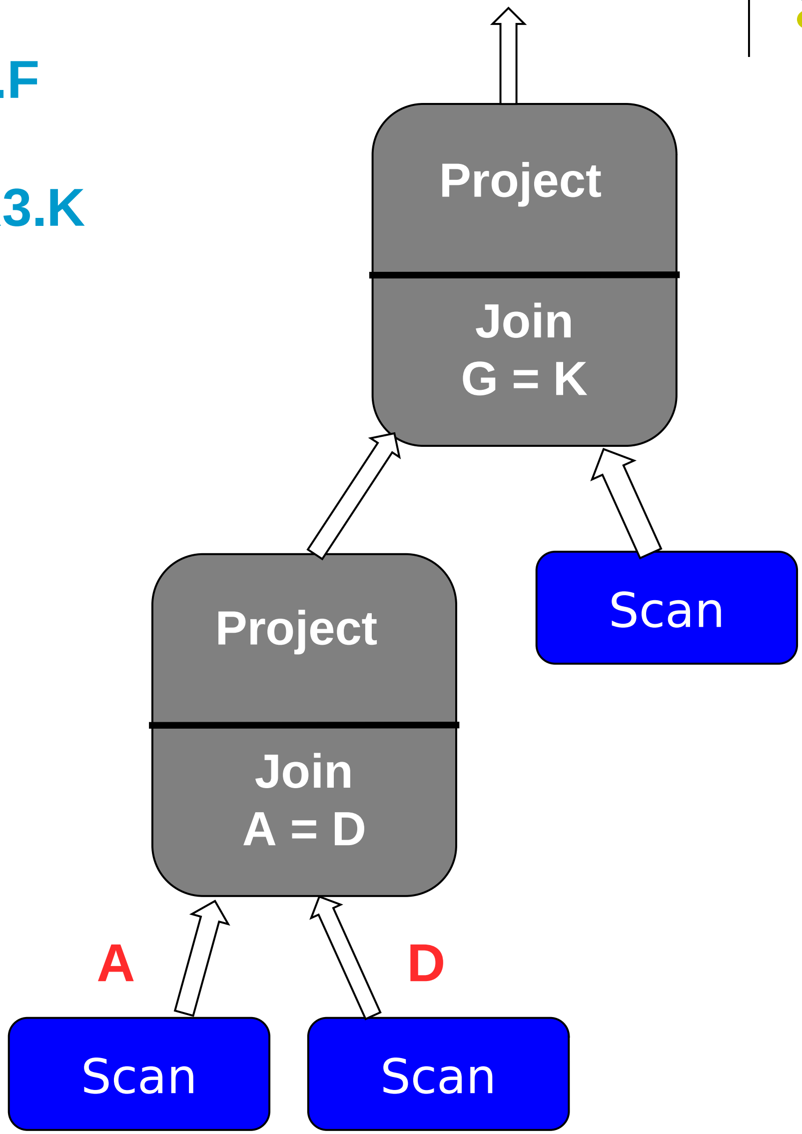




What about for plans with joins?

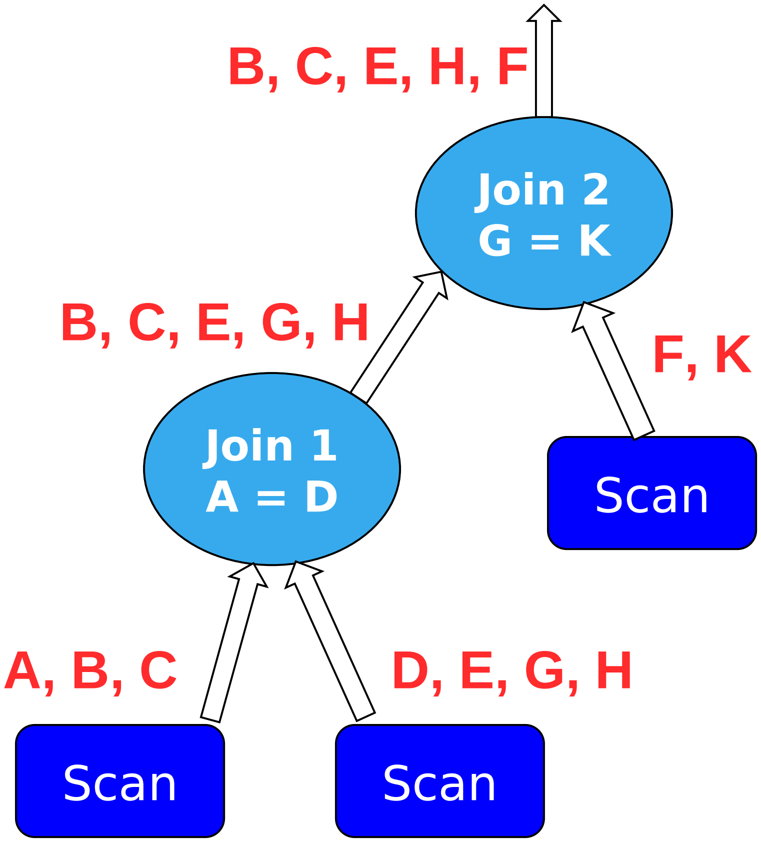
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K

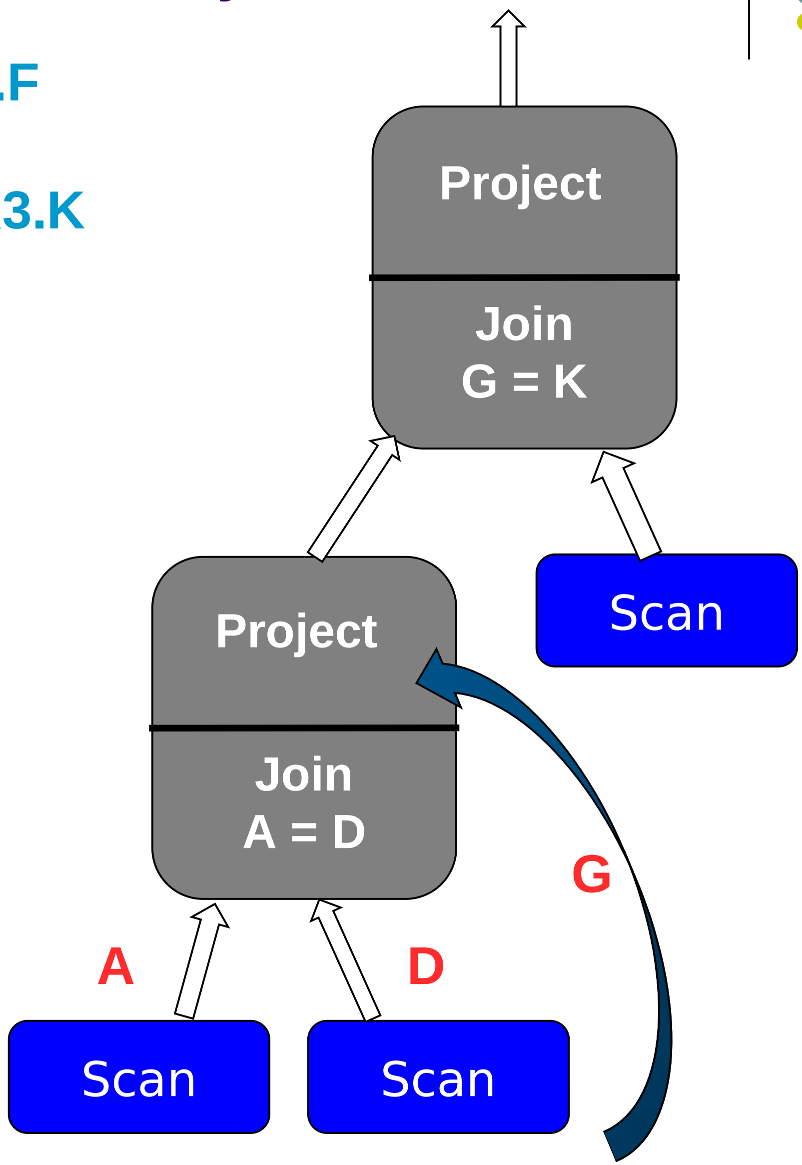




What about for plans with joins?

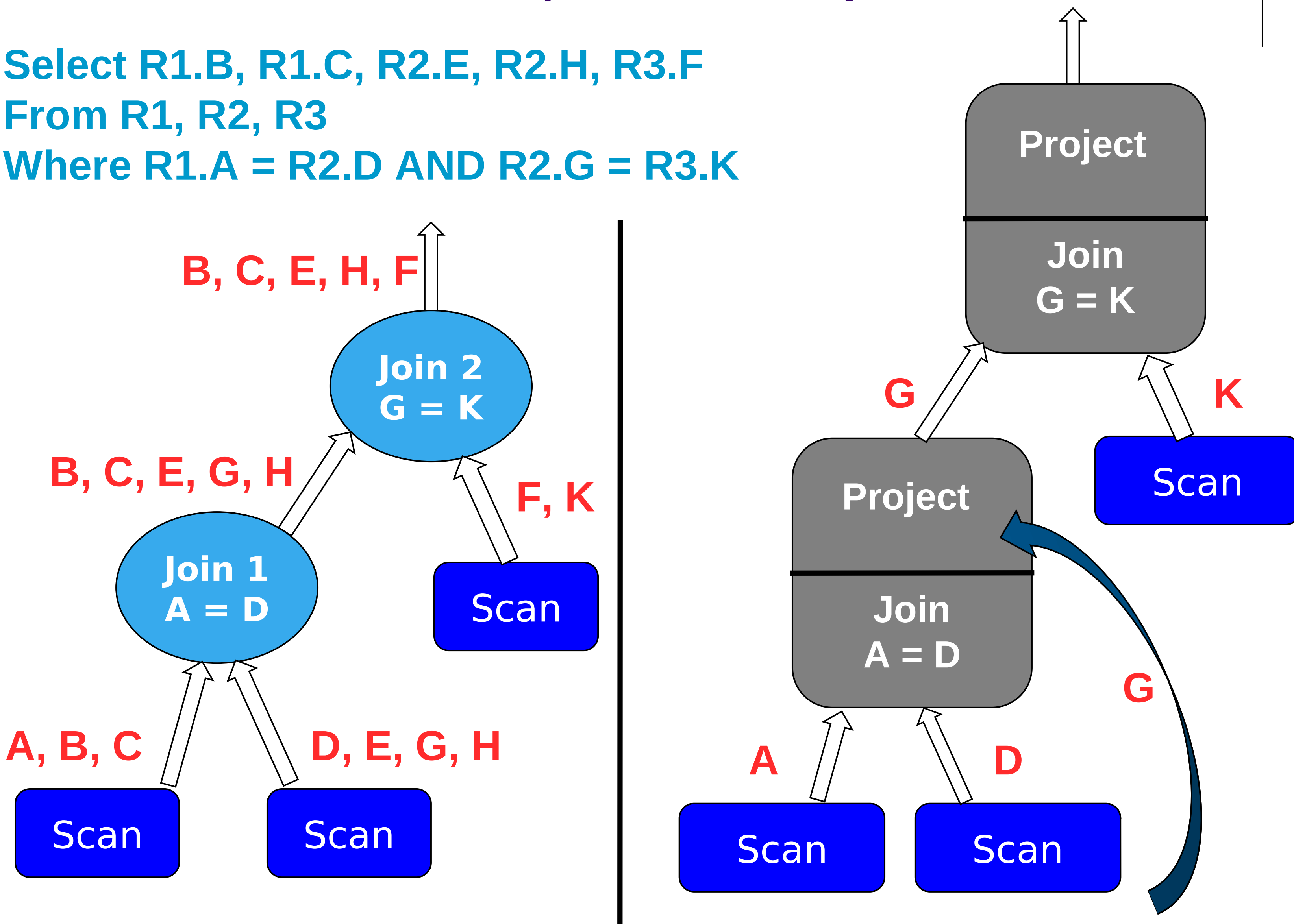
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K





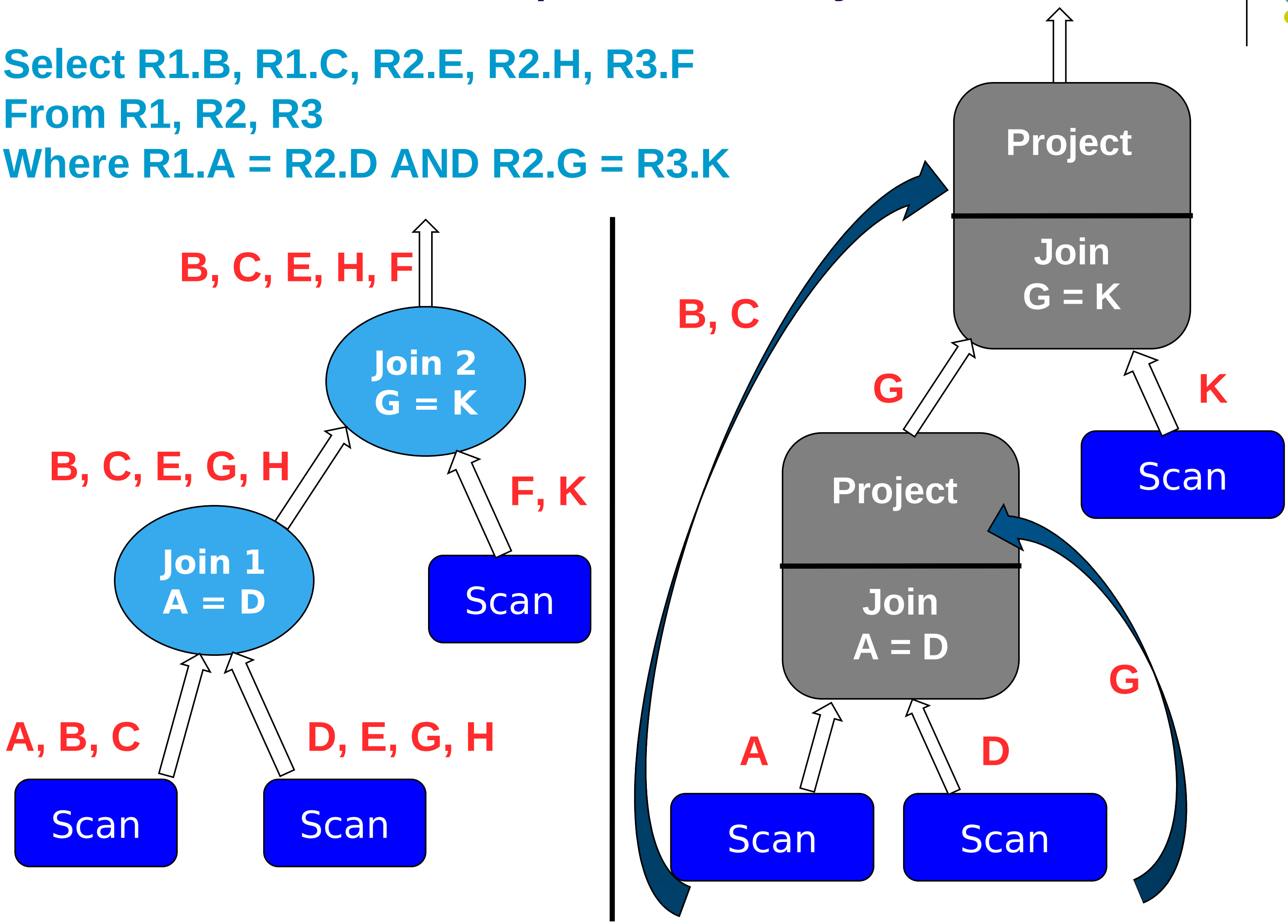
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



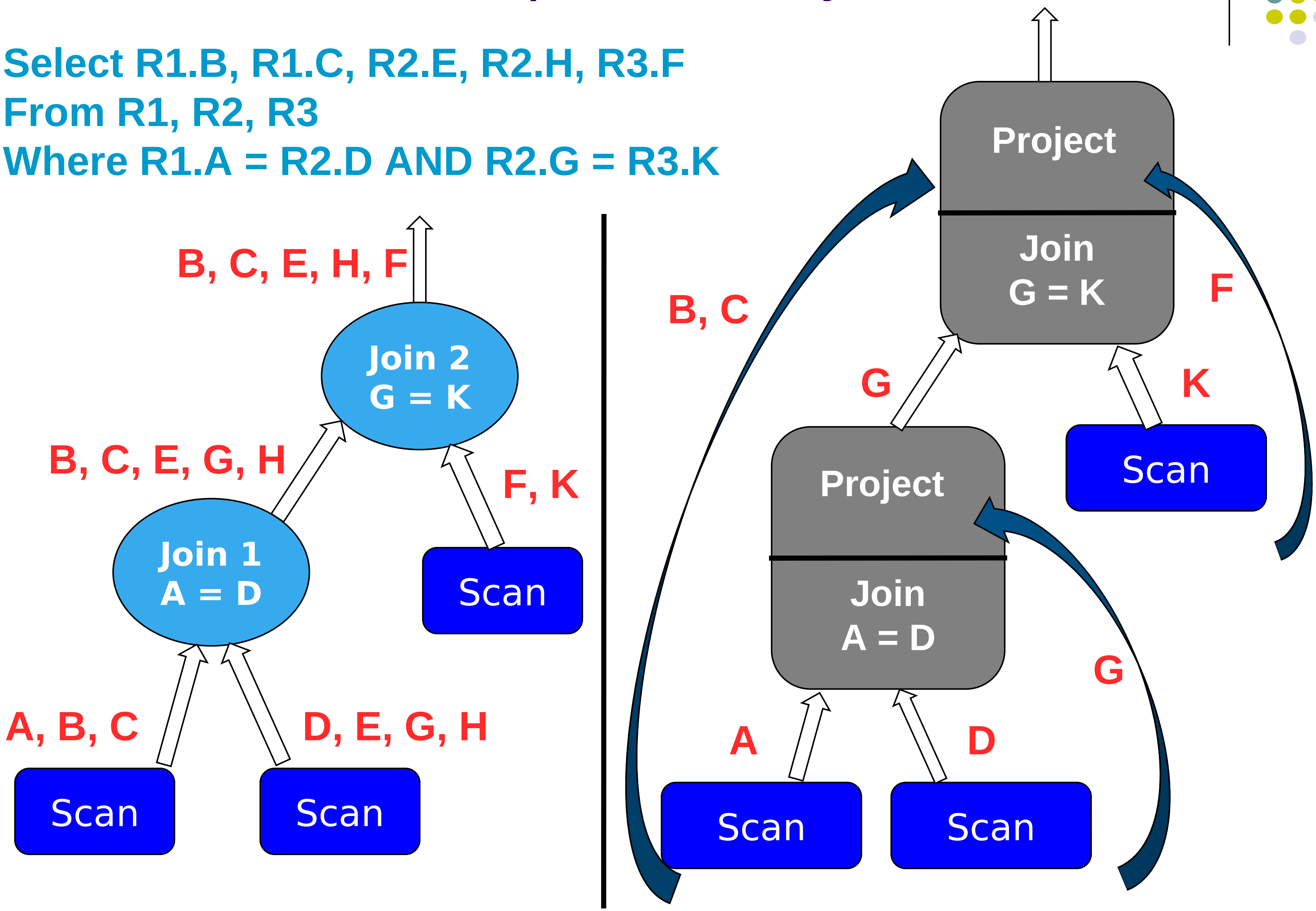
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



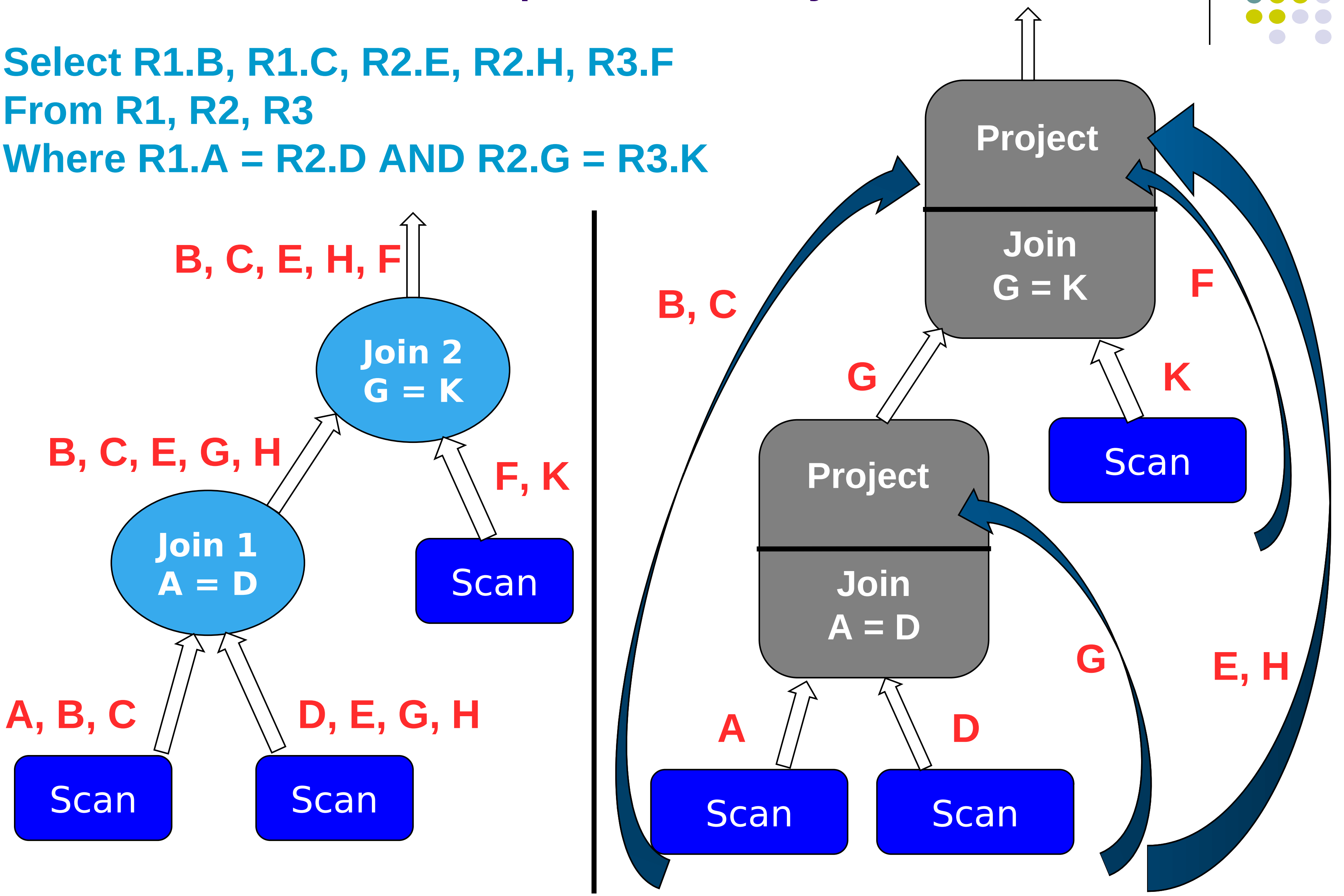
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



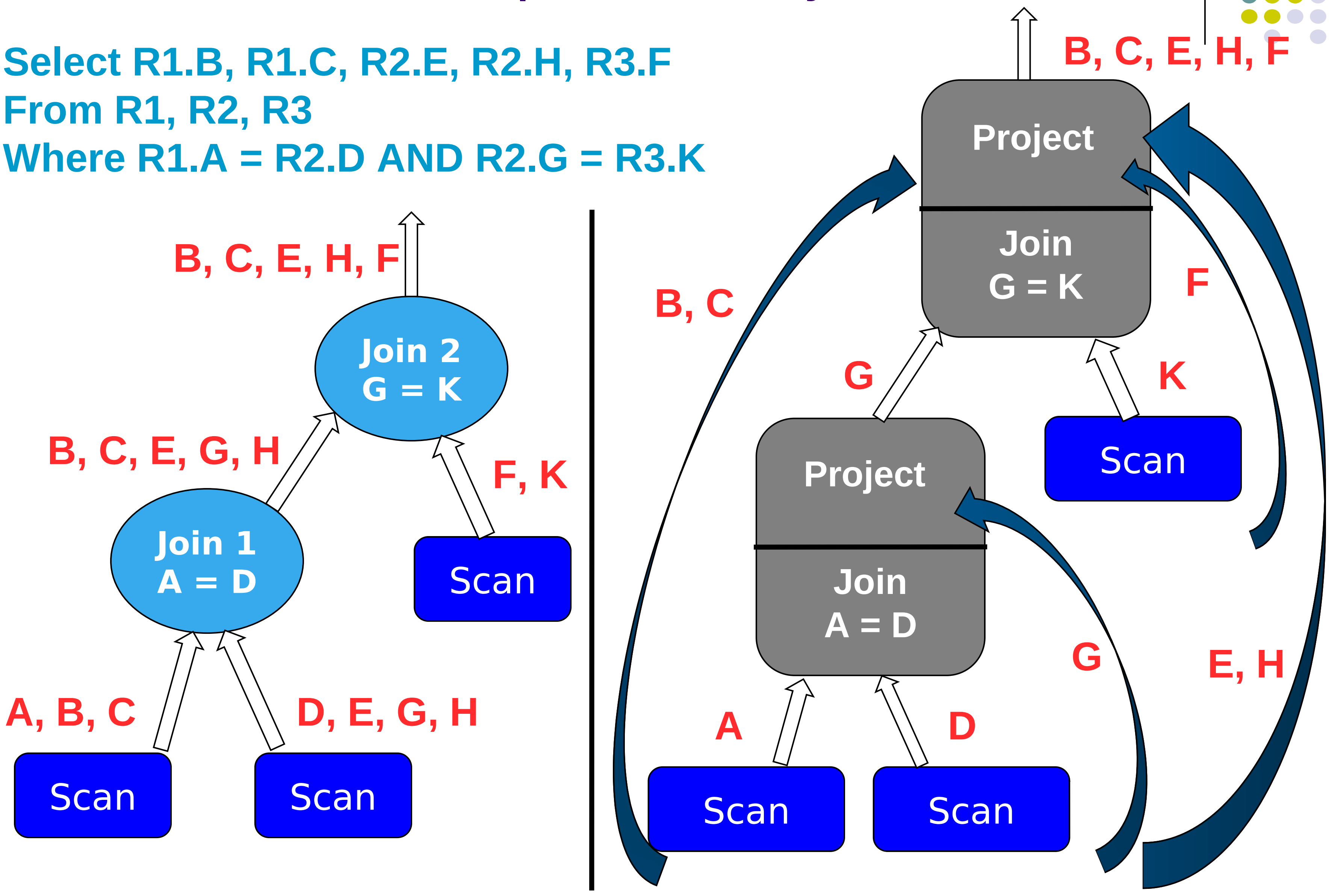
What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



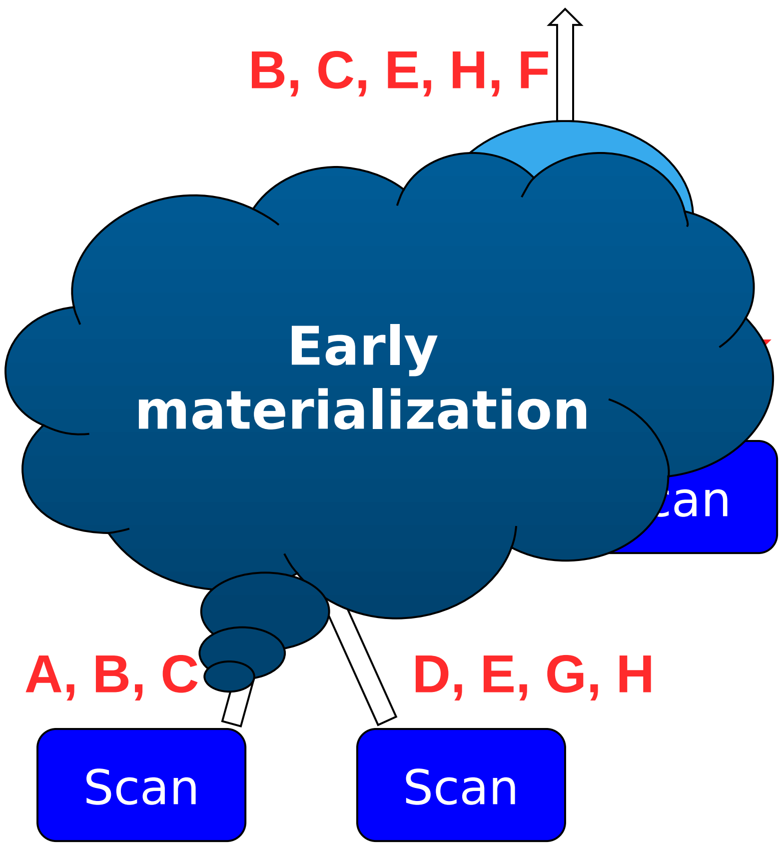
What about for plans with joins?

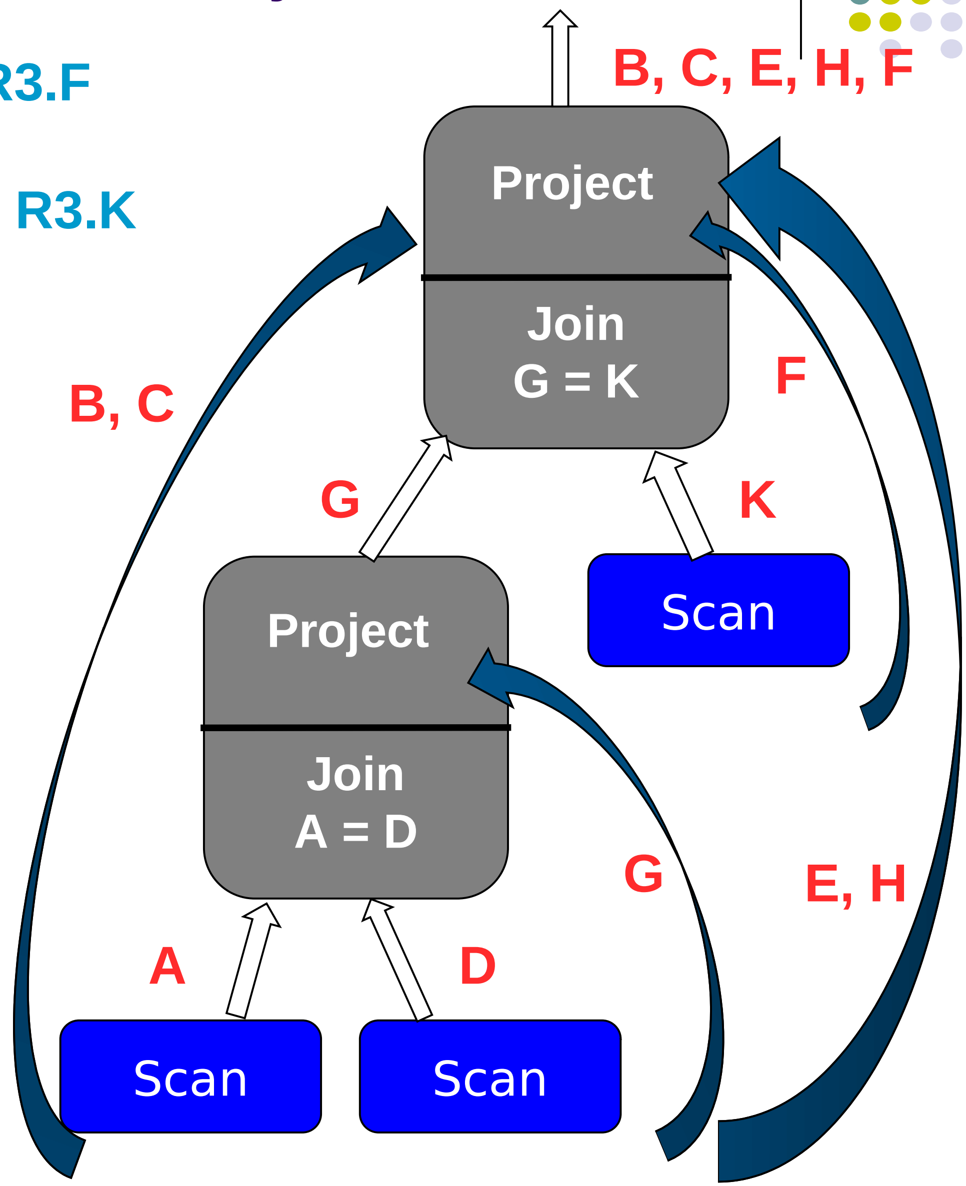
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



What about for plans with joins?

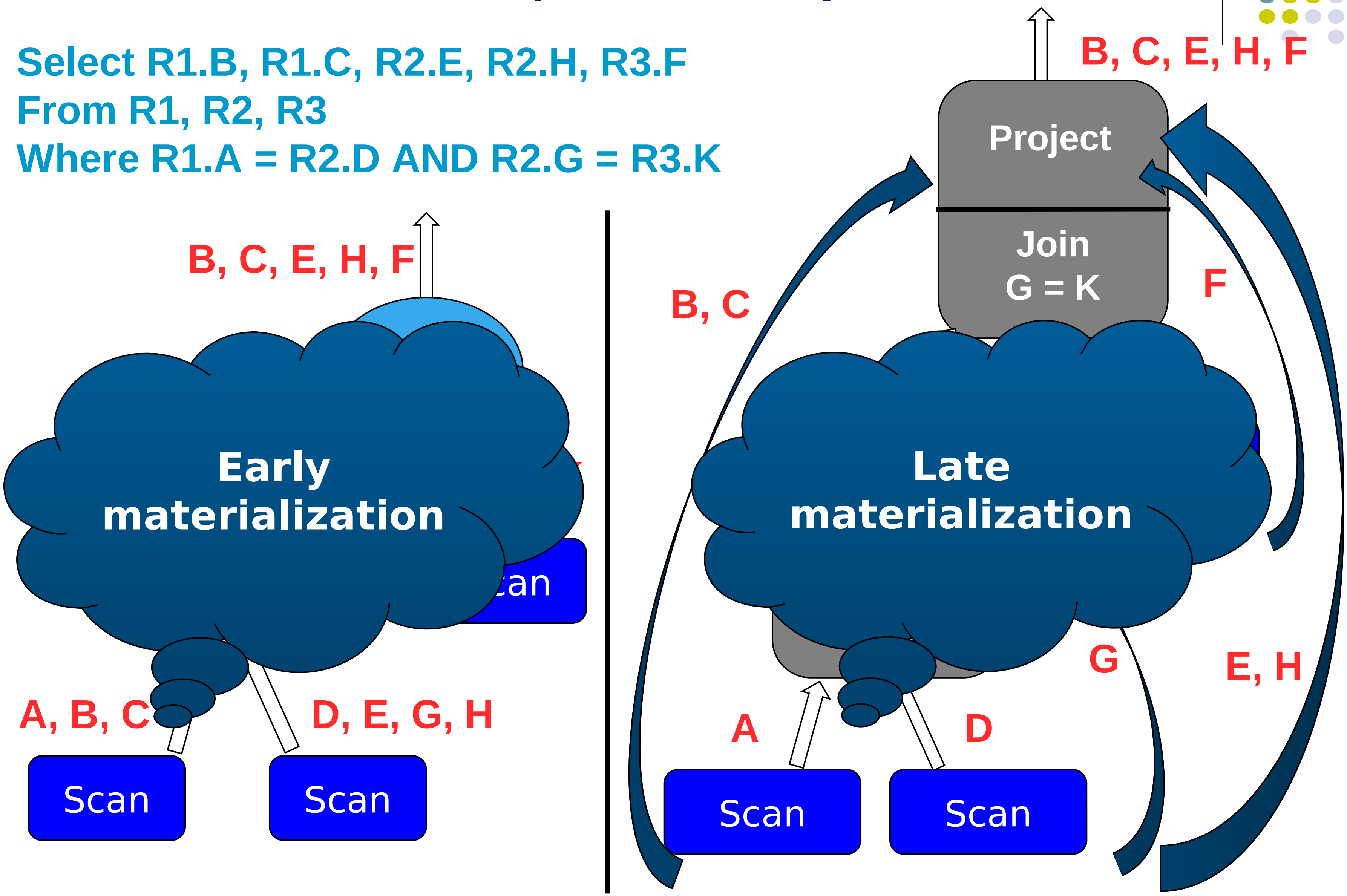
Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K





What about for plans with joins?

Select R1.B, R1.C, R2.E, R2.H, R3.F
From R1, R2, R3
Where R1.A = R2.D AND R2.G = R3.K



Early Materialization Example

QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

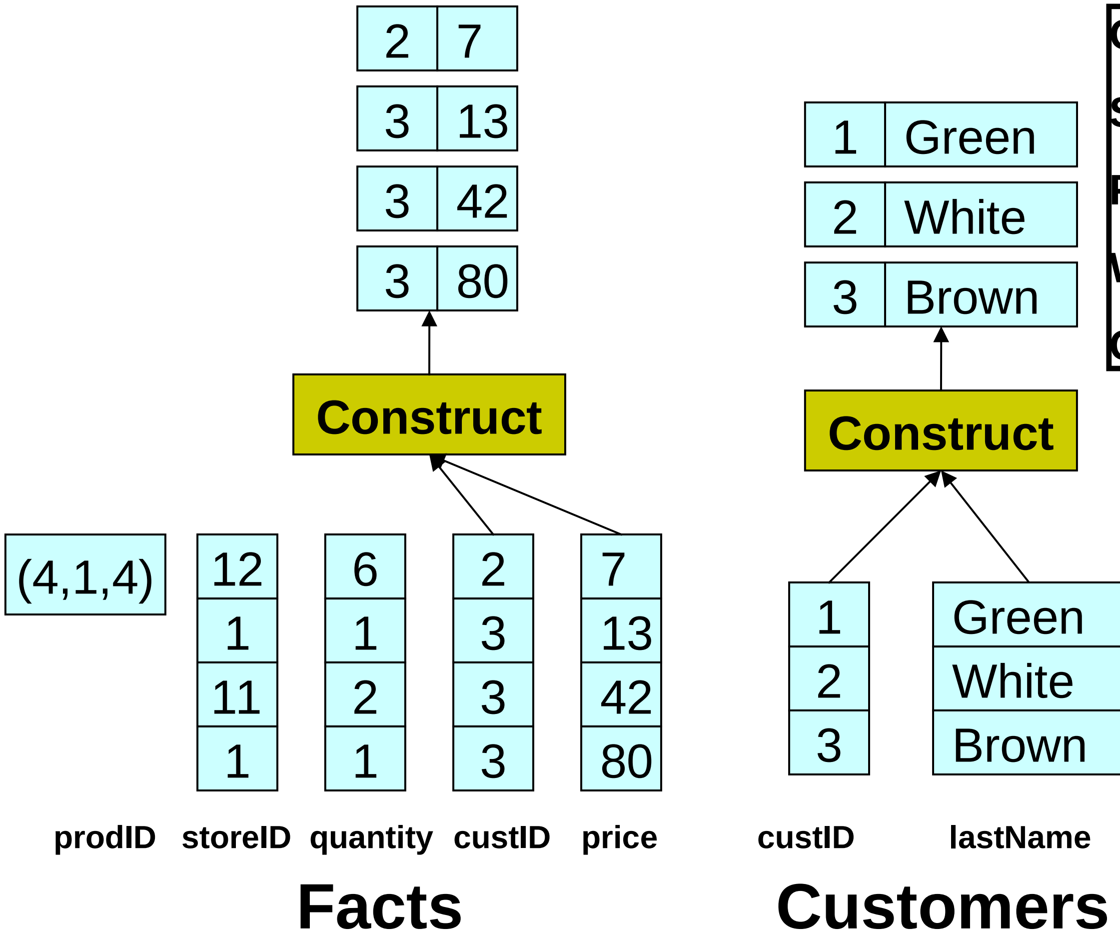
prodID	storeID	quantity	custID	price
(4,1,4)	12	6	2	7
	1	1	3	13
	11	2	3	42
	1	1	3	80

Facts

custID	lastName
1	Green
2	White
3	Brown

Customers

Early Materialization Example



QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

Early Materialization Example

2	7
3	13
3	42
3	80

1	Green
2	White
3	Brown

QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

Early Materialization Example

QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

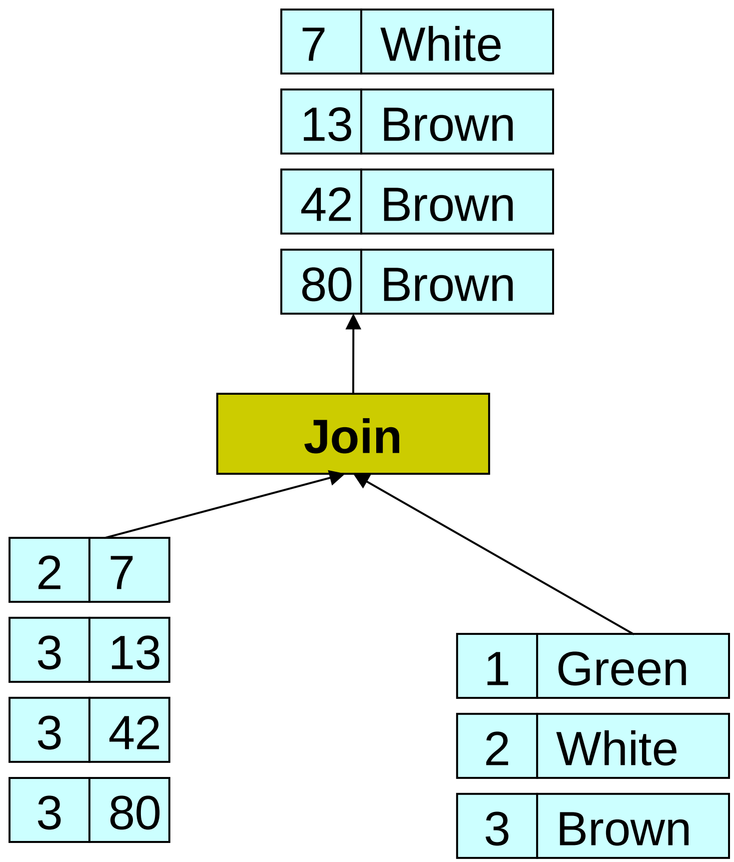
WHERE F.custID = C.custID

GROUP BY C.lastName

2	7
3	13
3	42
3	80

1	Green
2	White
3	Brown

Early Materialization Example



QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

Late Materialization Example

QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

prodID	storeID	quantity	custID	price
(4,1,4)	12	6	2	7
	1	1	3	13
	11	2	1	42
	1	1	3	80

Facts

custID	lastName
1	Green
2	White
3	Brown

Customers

Late Materialization Example

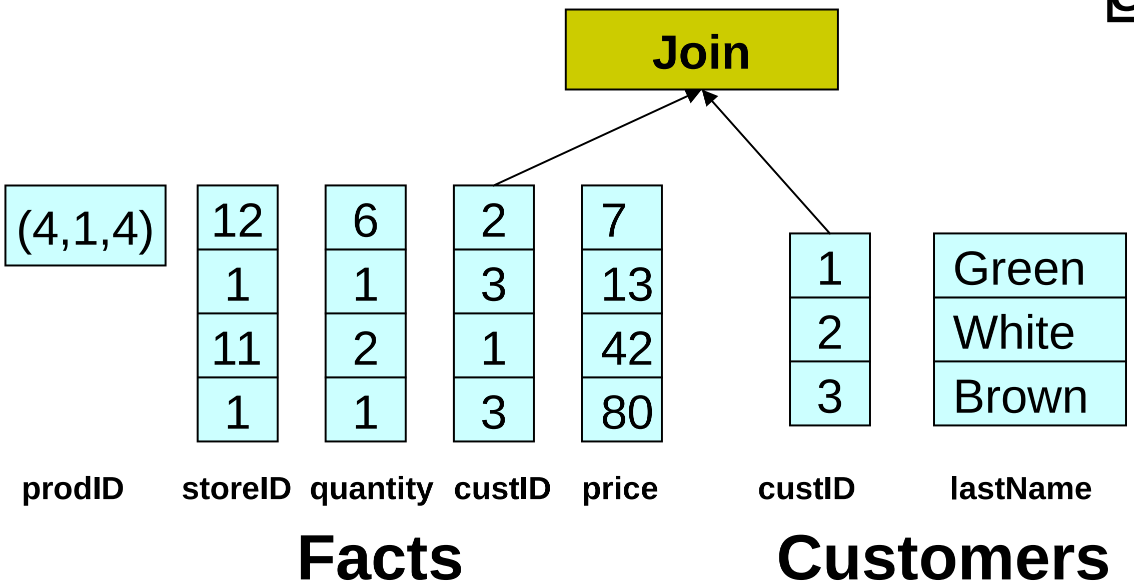
QUERY:

SELECT C.lastName,SUM(F.price)

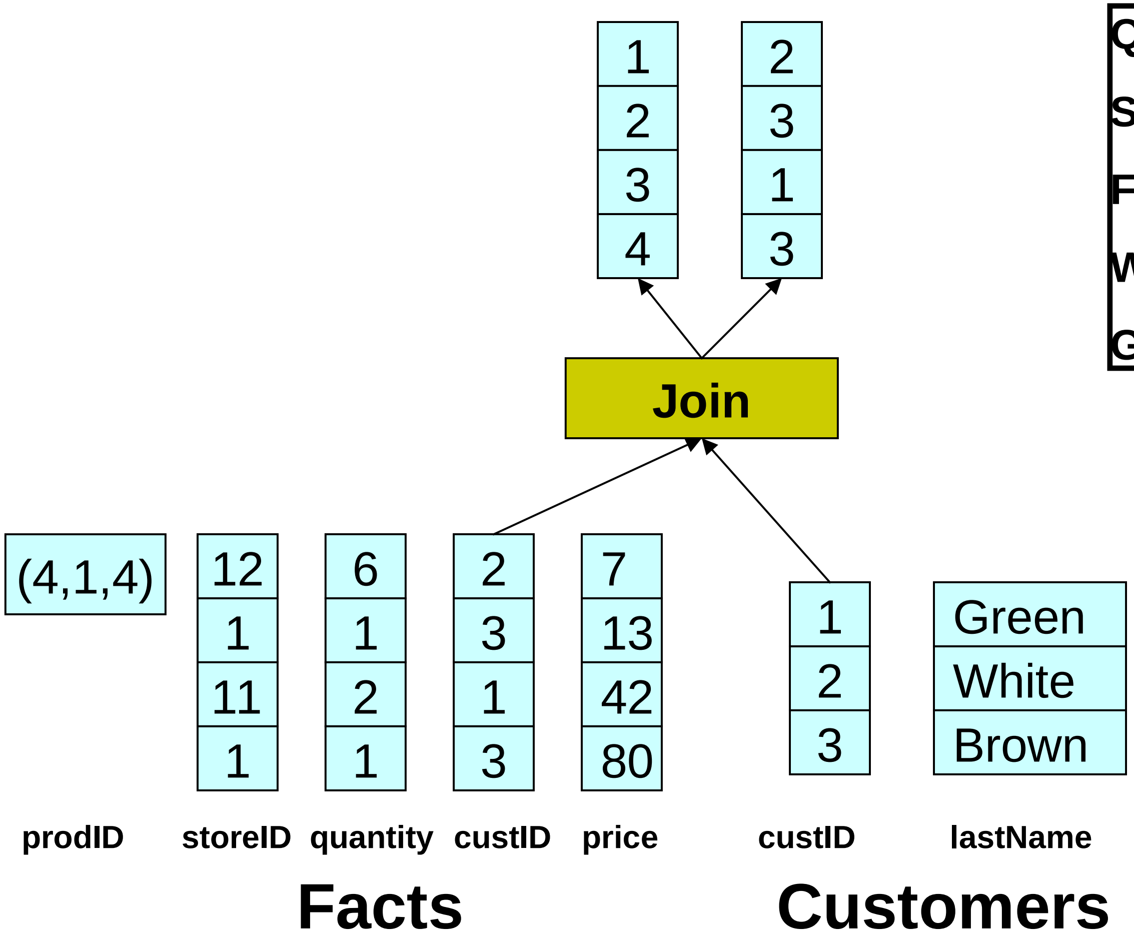
FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName



Late Materialization Example



QUERY:

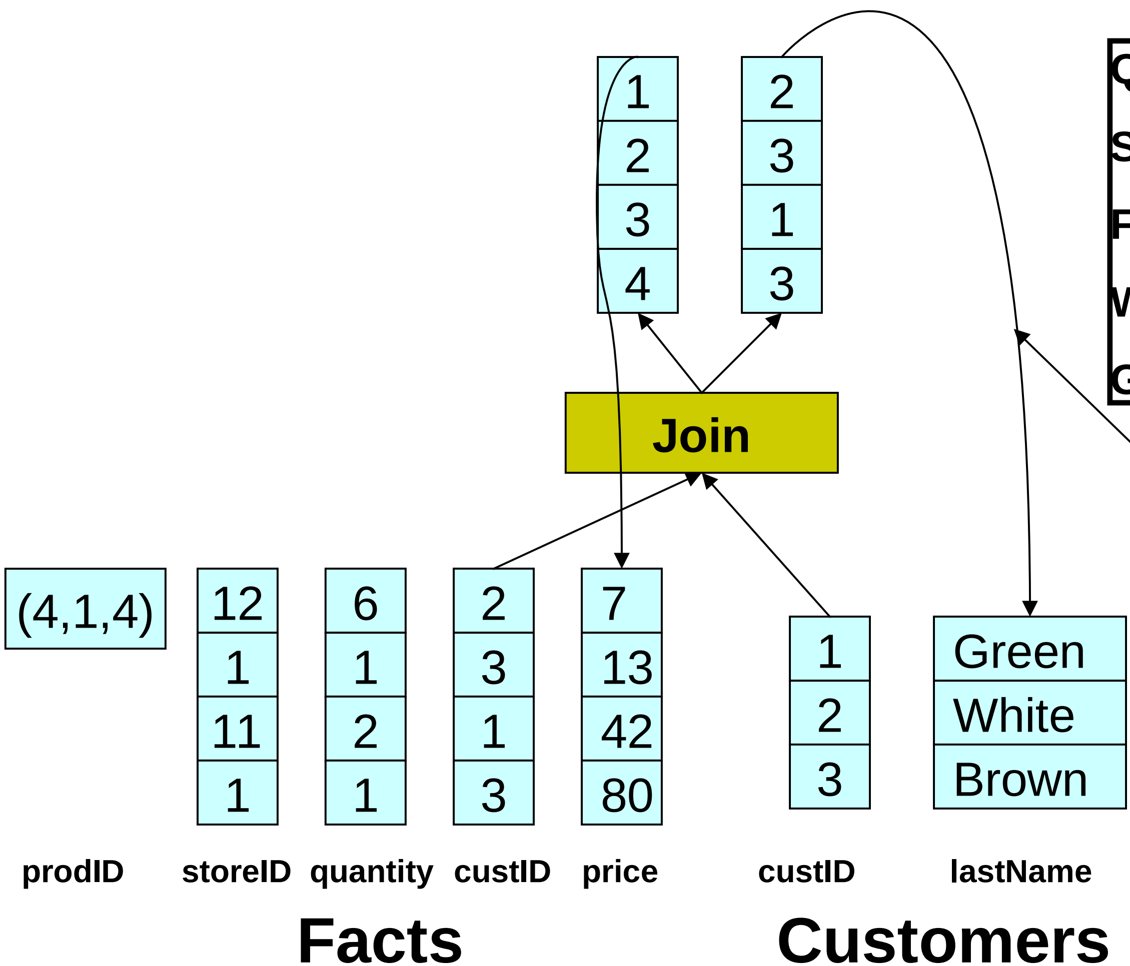
SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

Late Materialization Example



QUERY:

SELECT C.lastName,SUM(F.price)

FROM facts AS F, customers AS C

WHERE F.custID = C.custID

GROUP BY C.lastName

Late materialized join causes out of order probing of projected columns from the inner relation

Late Materialized Join Performance

- Naïve LM join (can be) about 2X slower than EM join on typical queries (due to random I/O)
  - This number is very dependent on
    - Amount of memory available
    - Number of projected attributes
    - Join cardinality
- But we can do better
  - Invisible Join
  - Jive/Flash Join
  - Radix cluster/decluster join

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

- Designed for typical joins when data is modeled using a star schema
  - One (big) "fact" table is joined with multiple (small) "dimension" tables

- Typical query (Star Schema Benchmark (SSBM)):

```
select c_nation, s_nation, d_year,
        sum(lo_revenue) as revenue
from lineorder, customer, supplier, date
where lo_custkey = c_custkey
   and lo_suppkey = s_suppkey
   and lo_orderdate = d_datekey
   and c_region = 'ASIA‘
   and s_region = 'ASIA‘
   and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

Apply "region = 'Asia'" On Customer Table

custkey	region	nation	…
1	ASIA	CHINA	…
2	ASIA	INDIA	…
3	ASIA	INDIA	…
4	EUROPE	FRANCE	…

→ **Hash Table (or bit-map) Containing Keys 1, 2 and 3**

Apply "region = 'Asia'" On Supplier Table

suppkey	region	nation	…
1	ASIA	RUSSIA	…
2	EUROPE	SPAIN	…
3	ASIA	JAPAN	…

→ **Hash Table (or bit-map) Containing Keys 1, 3**

Apply "year in [1992,1997]" On Date Table

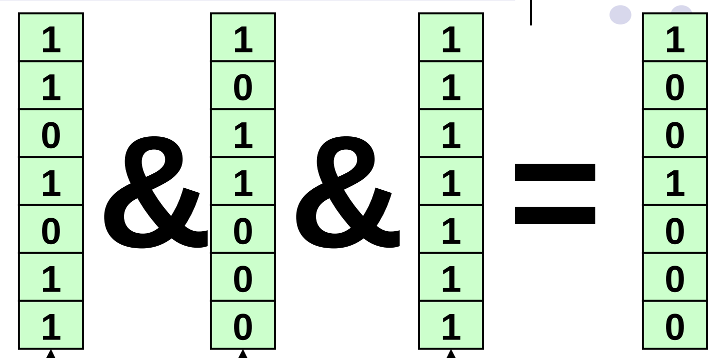
dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

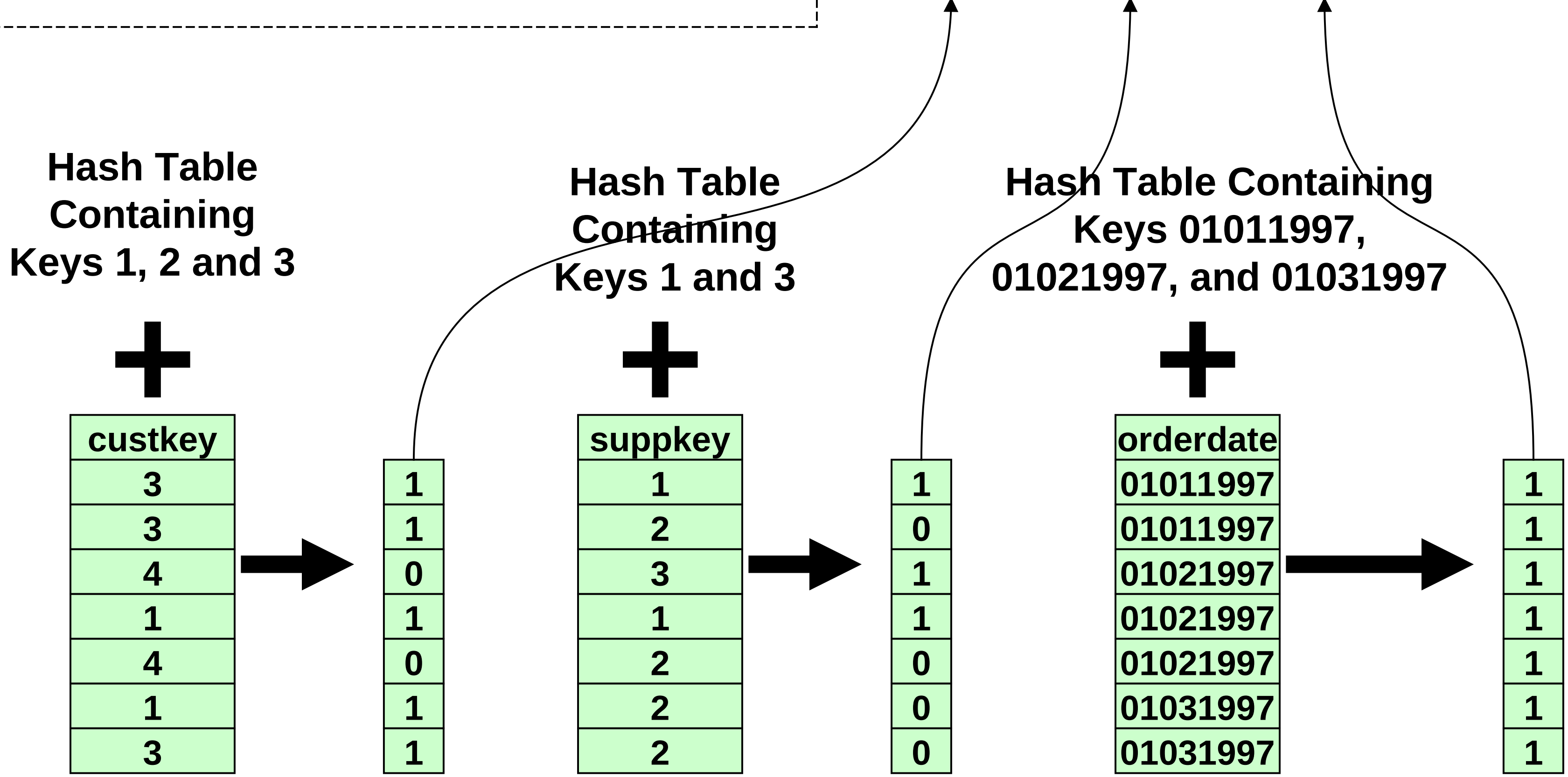
→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

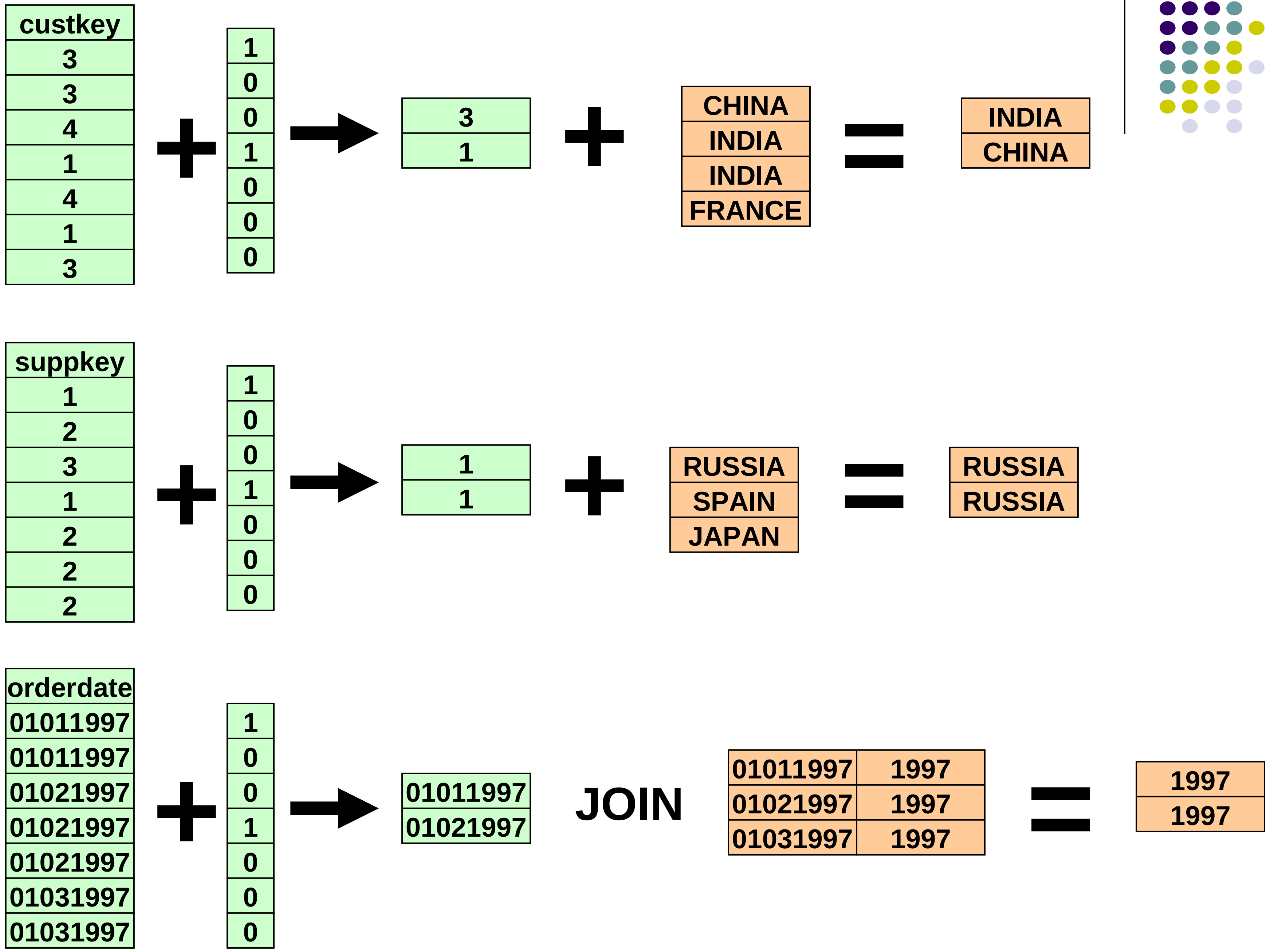
Original Fact Table

orderkey	custkey	suppkey	orderdate	revenue
1	3	1	01011997	43256
2	3	2	01011997	33333
3	4	3	01021997	12121
4	1	1	01021997	23233
5	4	2	01021997	45456
6	1	2	01031997	43251
7	3	2	01031997	34235

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi et. al. SIGMOD 2008.






custkey
3
3
4
1
4
1
3
1
0
0
1
0
0
0
3
1
CHINA
INDIA
INDIA
FRANCE
INDIA
CHINA
suppkey
1
2
3
1
2
2
2
1
0
0
1
0
0
0
1
1
RUSSIA
SPAIN
JAPAN
RUSSIA
RUSSIA
orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997
1
0
0
1
0
0
0
01011997
01021997
JOIN
01011997 1997
01021997 1997
01031997 1997
1997
1997

“Column-Stores vs Row-Stores: How Different are They Really?” Abadi, Madden, and Hachem. SIGMOD 2008.

Invisible Join

Apply “region = ‘Asia’” On Customer Table

custkey	region	nation	…
1	ASIA	CHINA	…
2	ASIA	INDIA	…
3	ASIA	INDIA	…
4	EUROPE	FRANCE	…

→ **Hash Table (or bit-map) Containing Keys 1, 2 and 3**

Apply “region = ‘Asia’” On Supplier Table

suppkey	region	nation	…
1	ASIA	RUSSIA	…
2	EUROPE	SPAIN	…
3	ASIA	JAPAN	…

→ **Hash Table (or bit-map) Containing Keys 1, 3**

Apply “year in [1992,1997]” On Date Table

dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

Invisible Join

"Column-Stores vs Row-Stores: How Different are They Really?" Abadi, Madden, and Hachem. SIGMOD 2008.

Apply "region = 'Asia'" On Customer Table

custkey	region	nation	…
1	ASIA	CHINA	…
2	ASIA	INDIA	…
3	ASIA	INDIA	…
4	EUROPE	FRANCE	…

→ ~~**Hash Table (or bit-map) Containing Keys 1, 2 and 3**~~

Range [1-3]
(between-predicate rewriting)

Apply "region = 'Asia'" On Supplier Table

suppkey	region	nation	…
1	ASIA	RUSSIA	…
2	EUROPE	SPAIN	…
3	ASIA	JAPAN	…

→ **Hash Table (or bit-map) Containing Keys 1, 3**

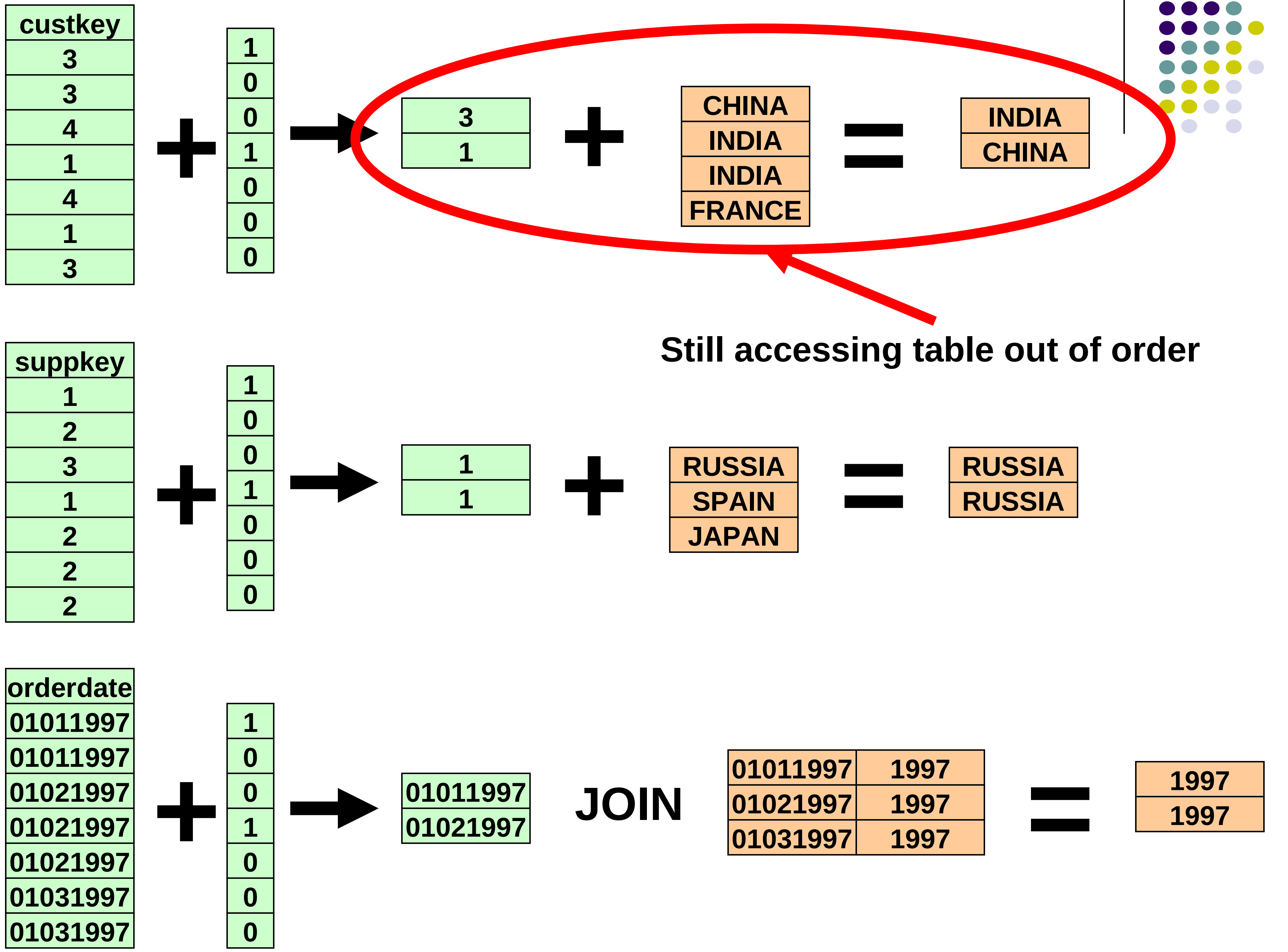
Apply "year in [1992,1997]" On Date Table

dateid	year	…
01011997	1997	…
01021997	1997	…
01031997	1997	…

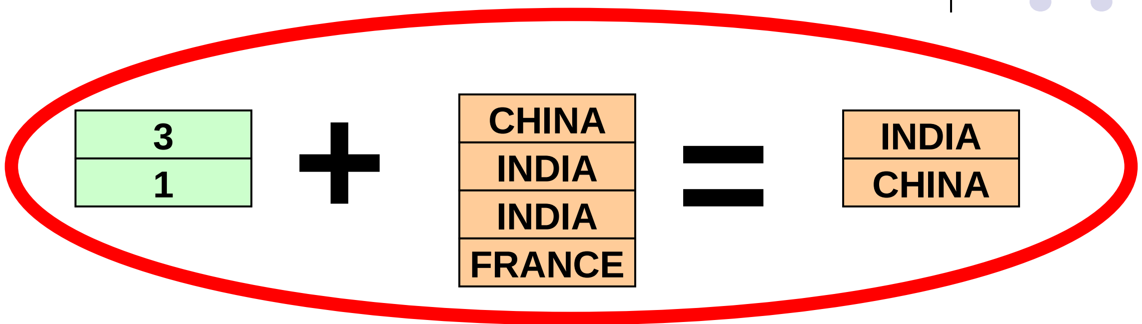
→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

Invisible Join

- **Bottom Line**
  - **Many data warehouses model data using star/snowflake schemes**
  - **Joins of one (fact) table with many dimension tables is common**
  - **Invisible join takes advantage of this by making sure that the table that can be accessed in position order is the fact table for each join**
  - **Position lists from the fact table are then intersected (in position order)**
  - **This reduces the amount of data that must be accessed out of order from the dimension tables**
  - **“Between-predicate rewriting” trick not relevant for this discussion**


custkey
3
3
4
1
4
1
3
1
0
0
1
0
0
0
3
1
CHINA
INDIA
INDIA
FRANCE
INDIA
CHINA
Still accessing table out of order
suppkey
1
2
3
1
2
2
2
1
0
0
1
0
0
0
1
1
RUSSIA
SPAIN
JAPAN
RUSSIA
RUSSIA
orderdate
01011997
01011997
01021997
01021997
01021997
01031997
01031997
1
0
0
1
0
0
0
01011997
01021997
JOIN
01011997 1997
01021997 1997
01031997 1997
1997
1997

Jive/Flash Join

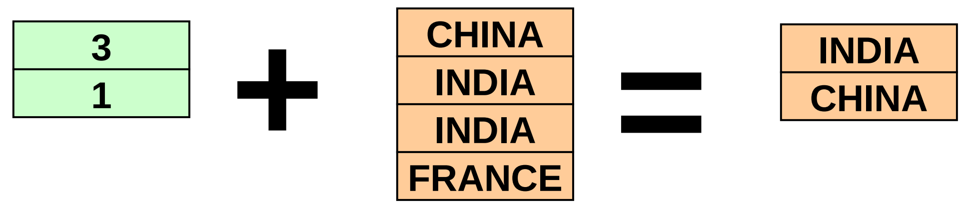


Still accessing table out of order

"Fast Joins using Join Indices". Li and Ross, VLDBJ 8:1-24, 1999.

"Query Processing Techniques for Solid State Drives". Tsirogiannis, Harizopoulos et. al. SIGMOD 2009.

Jive/Flash Join



Jive/Flash Join

1. Add column with dense ascending integers from 1

1	**3**
2	**1**

+

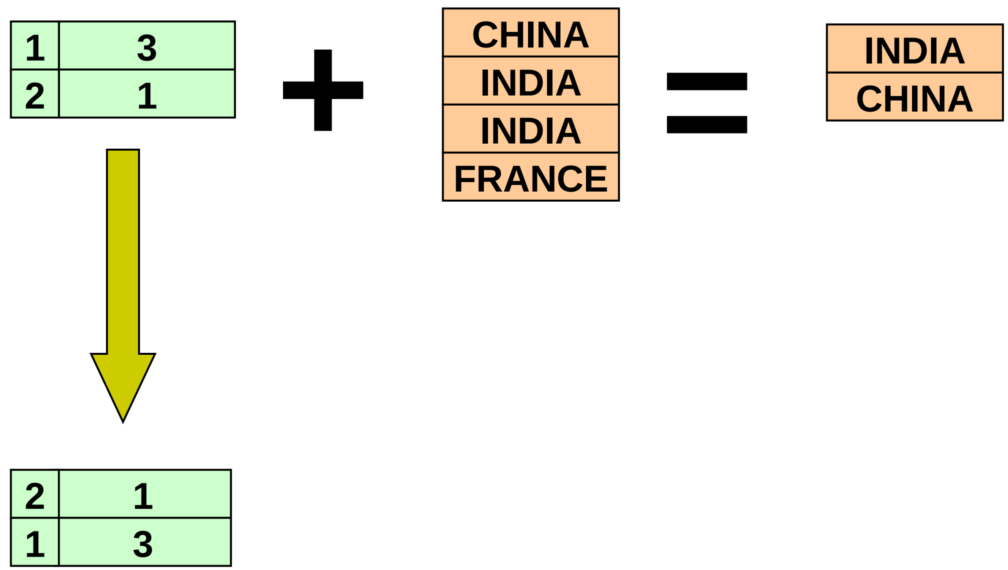
CHINA
INDIA
INDIA
FRANCE

=

INDIA
CHINA

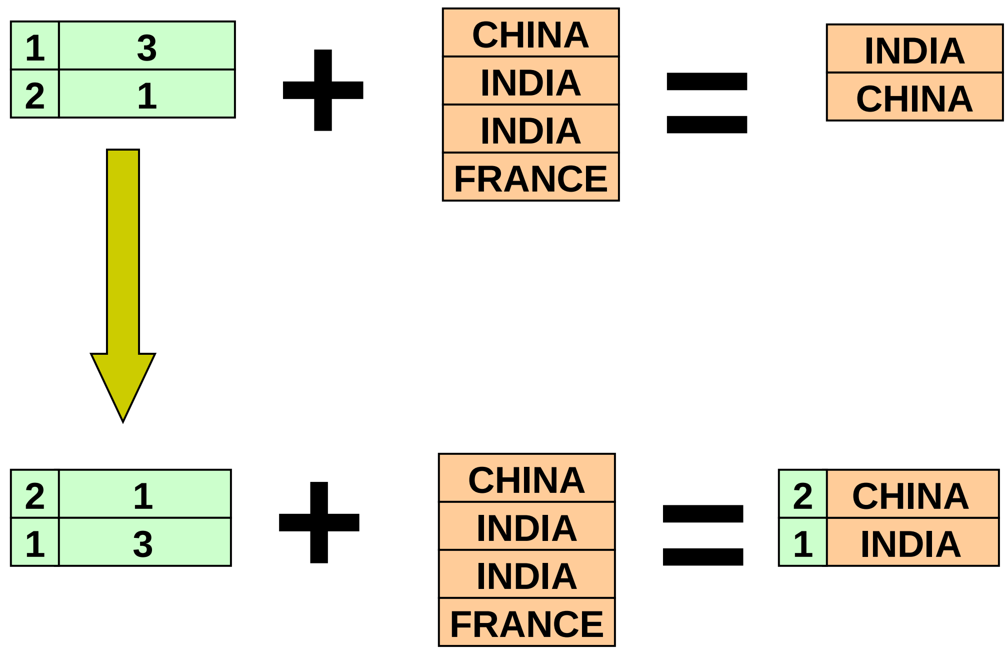
Jive/Flash Join

1. Add column with dense ascending integers from 1
2. Sort new position list by second column



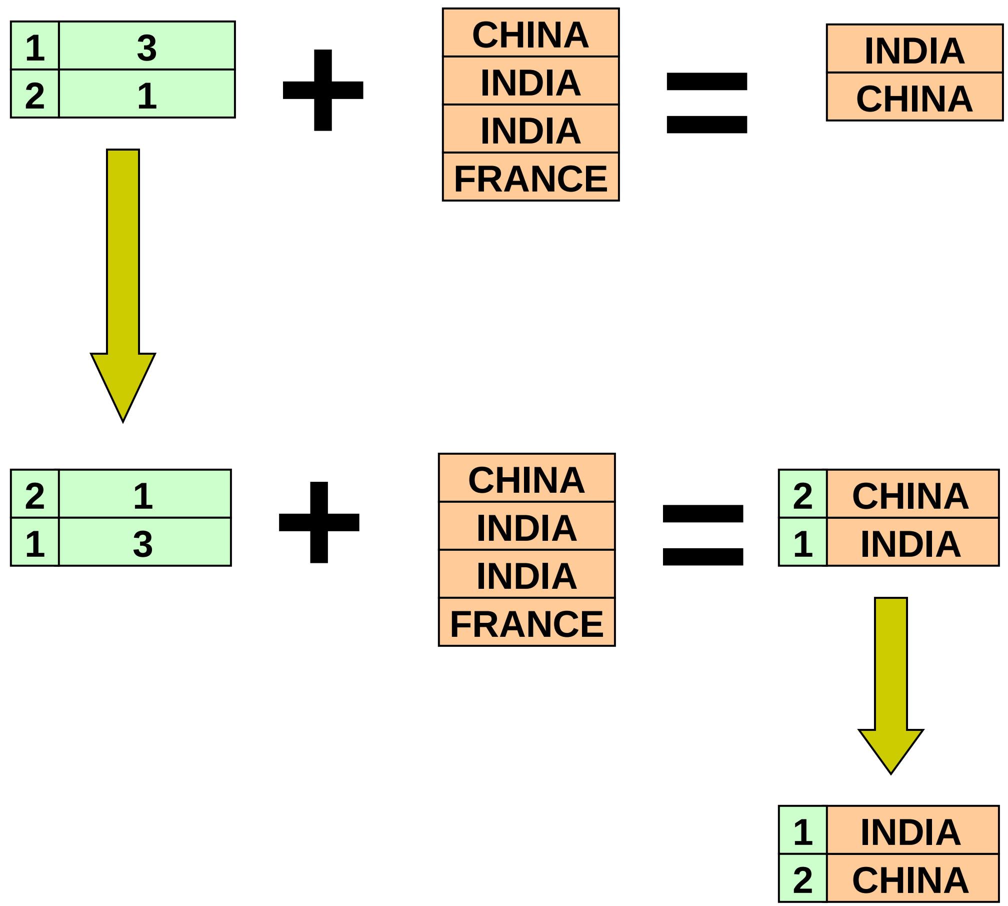
Jive/Flash Join

1. Add column with dense ascending integers from 1
2. Sort new position list by second column
3. Probe projected column in order using new sorted position list, keeping first column from position list around



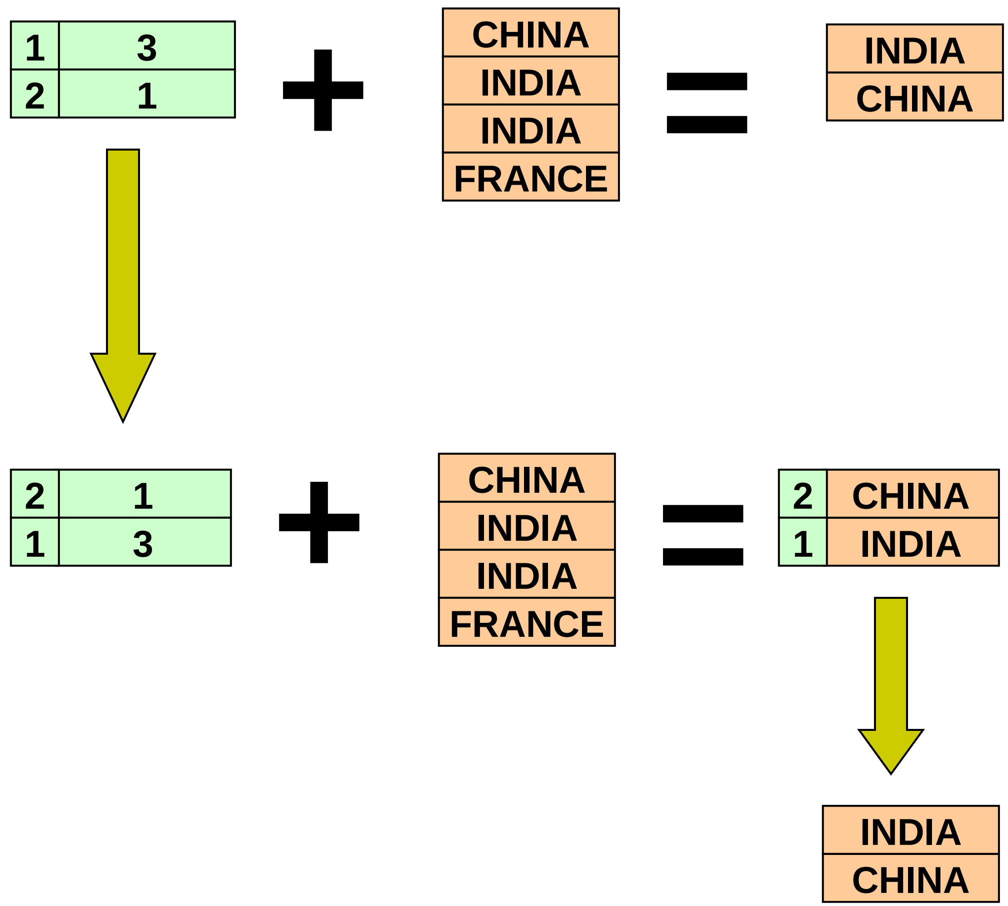
Jive/Flash Join

1. Add column with dense ascending integers from 1
2. Sort new position list by second column
3. Probe projected column in order using new sorted position list, keeping first column from position list around
4. Sort new result by first column



Jive/Flash Join

1. Add column with dense ascending integers from 1
2. Sort new position list by second column
3. Probe projected column in order using new sorted position list, keeping first column from position list around
4. Sort new result by first column
5. Remove first column



Jive/Flash Join

- **Bottom Line**
  - **Instead of probing projected columns from inner table out of order:**
    - **Sort join index**
    - **Probe projected columns in order**
    - **Sort result using an added column**
  - **LM vs EM tradeoffs:**
    - **LM has the extra sorts (EM accesses all columns in order)**
    - **LM only has to fit join columns into memory (EM needs join columns and all projected columns)**
      - **Results in big memory and CPU savings (see part 3 for why there is CPU savings)**
    - **LM only has to materialize relevant columns**
    - **In many cases LM advantages outweigh disadvantages**
  - **LM would be a clear winner if not for those pesky sorts … can we do better?**

ADM: Literature

- **Column-Oriented Database Systems (2/6) - Selected Execution Techniques**
  - Compression
    - “Compressing Relations and Indexes”. Goldstein, Ramakrishnan, Shaft. ICDE’98.
    - “Query optimization in compressed database systems”. Chen, Gehrke, Korn. SIGMOD’01.
    - “Super-Scalar RAM-CPU Cache Compression”. Zukowski, Heman, Nes, Boncz. ICDE’06.
    - “Integrating Compression and Execution in Column-Oriented Database Systems”. Abadi, Madden, Ferreira. SIGMOD’06.
    - “Improved Word-Aligned Binary Compression for Text Indexing”. Ahn, Moffat. TKDE’06.
  - Tuple Materialization
    - “Materialization Strategies in a Column-Oriented DBMS”. Abadi, Myers, DeWitt, Madden. ICDE’07.
    - “Column-Stores vs Row-Stores: How Different are They Really?”. Abadi, Madden, Hachem. SIGMOD’08.
    - “Query Processing Techniques for Solid State Drives”. Tsirogiannis, Harizopoulos Shah, Wiener, Graefe. SIGMOD’09.
    - “Self-organizing tuple reconstruction in column-stores“. Idreos, Manegold, Kersten. SIGMOD’09.
  - Join
    - “Fast Joins using Join Indices”. Li and Ross. VLDBJ 8:1-24, 1999.

ADM: Agenda

- 07.09.2022: Lecture 1: **Introduction**
- 14.09.2022: Lecture 2: **SQL Recap**

  (plus Assignment 1 [in groups; 3 weeks]: TPC-H benchmark)
- 21.09.2022: Lecture 3: **Column-Oriented Database Systems (1/6) - Motivation & Basic Concepts**
- 28.09.2022: Lecture 4: **Column-Oriented Database Systems (2a/6) - Selected Execution Techniques (1/2)**
- 05.10.2022: Lecture 5: **Column-Oriented Database Systems (2b/6) - Selected Execution Techniques (2/2)**

  (plus Assignment 2 [in groups; 3 weeks]: Compression techniques)
- 12.10.2022: Lecture 6: **Column-Oriented Database Systems (3/6) - Cache Conscious Joins**
- 19.10.2022: Lecture 7: **Column-Oriented Database Systems (4/6) - "Vectorized Execution"**
- ~~26.10.2022~~: ***No lecture!***
- 02.11.2022: Lecture 8: **DuckDB: An embedded database for data science (1/2) (guest lecture & *hands-on*)**

  (plus Assignment 3 [individual; 2 weeks]: Analysing NYC Cab dataset with DuckDB)
- 09.11.2022: Lecture 9: **DuckDB: An embedded database for data science (2/2) (guest lecture & *hands-on*)**
- 16.11.2022: Lecture 10: **Branch Misprediction & Predication**

  (plus Assignment 4 [individual; 2 weeks]: Predication)
- 23.11.2022: Lecture 11: **Column-Oriented Database Systems (5/6) - Adaptive Indexing**
- 30.11.2022: Lecture 12: **Column-Oriented Database Systems (6/6) - Progressive Indexing**

ADM: Literature

- **Column-Oriented Database Systems (3/6) - Cache Conscious Joins**
  - “Cache Conscious Algorithms for Relational Query Processing”. Shatdal, Kant, Naughton. VLDB’94.
  - “Fast Joins using Join Indices”. Li and Ross. VLDBJ 8:1-24, 1999.
  - “Optimizing main-memory join on modern hardware”. Boncz, Manegold, Kersten, TKDE 14(4): 709-730, 2002.
    - “Database Architecture Optimized for the New Bottleneck: Memory Access”. Boncz, Manegold, Kersten. VLDB’99.
    - “What Happens During a Join? Dissecting CPU and Memory Optimization Effects”. Manegold, Boncz, Kersten. VLDB’00.
    - “Optimizing database architecture for the new bottleneck: memory access”. Manegold, Boncz, Kersten. VLDB J. 9(3): 231-246, 2000.
    - “Generic Database Cost Models for Hierarchical Memory Systems”. Manegold, Boncz, Kersten. VLDB’02.
  - “Cache-Conscious Radix-Decluster Projections”. Manegold, Boncz, Nes. VLDB’04.