Query Execution in Column-Stores

Atte Hinkka

Seminar on Columnar Databases, Fall 2012

Central concepts

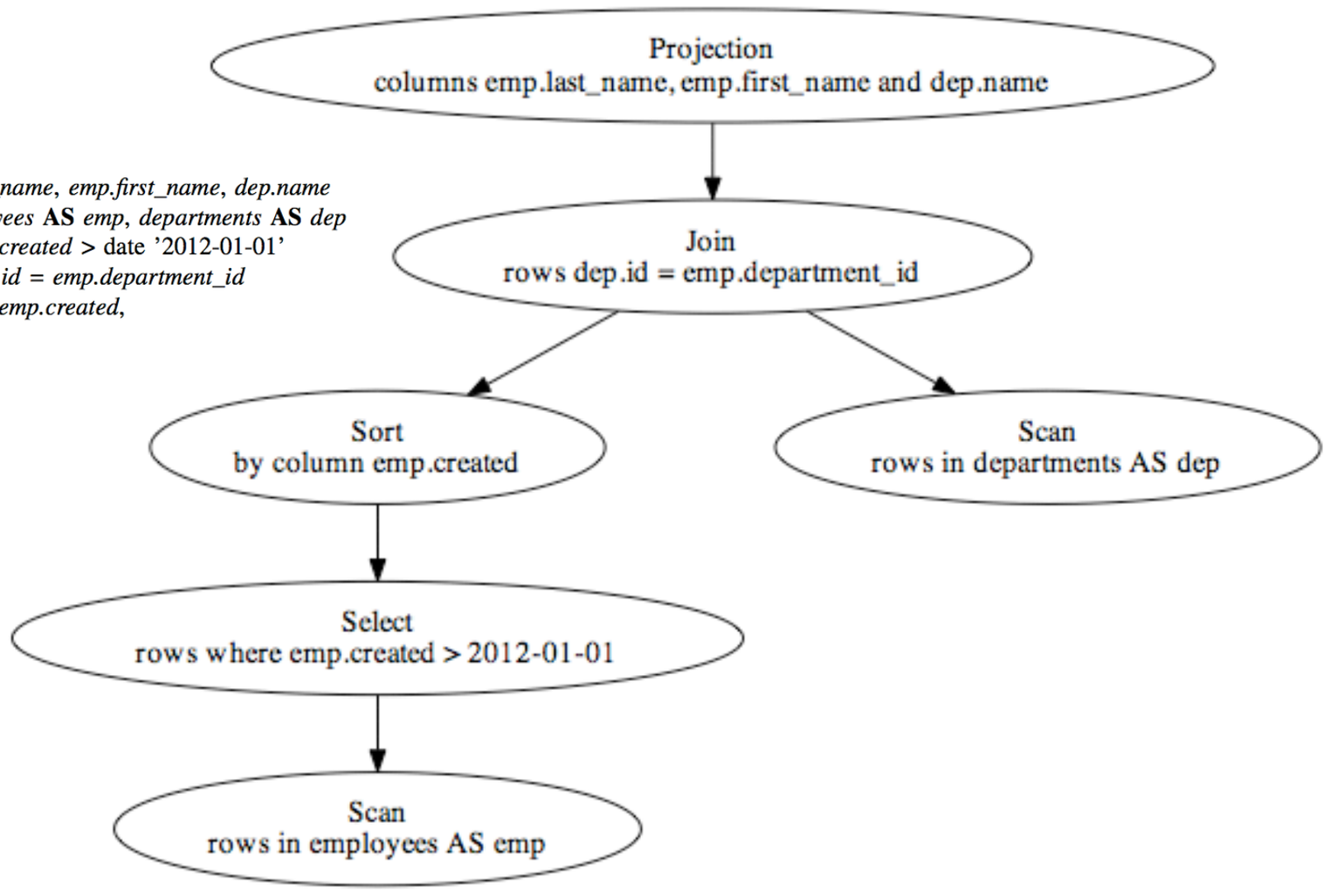
- Column (query) operators
- Compression considerations
- Materialization strategies
- Vectorized operations

Query what?

- Query operator trees
- Models for query execution
- Architectural models
- Roots in the transactional world

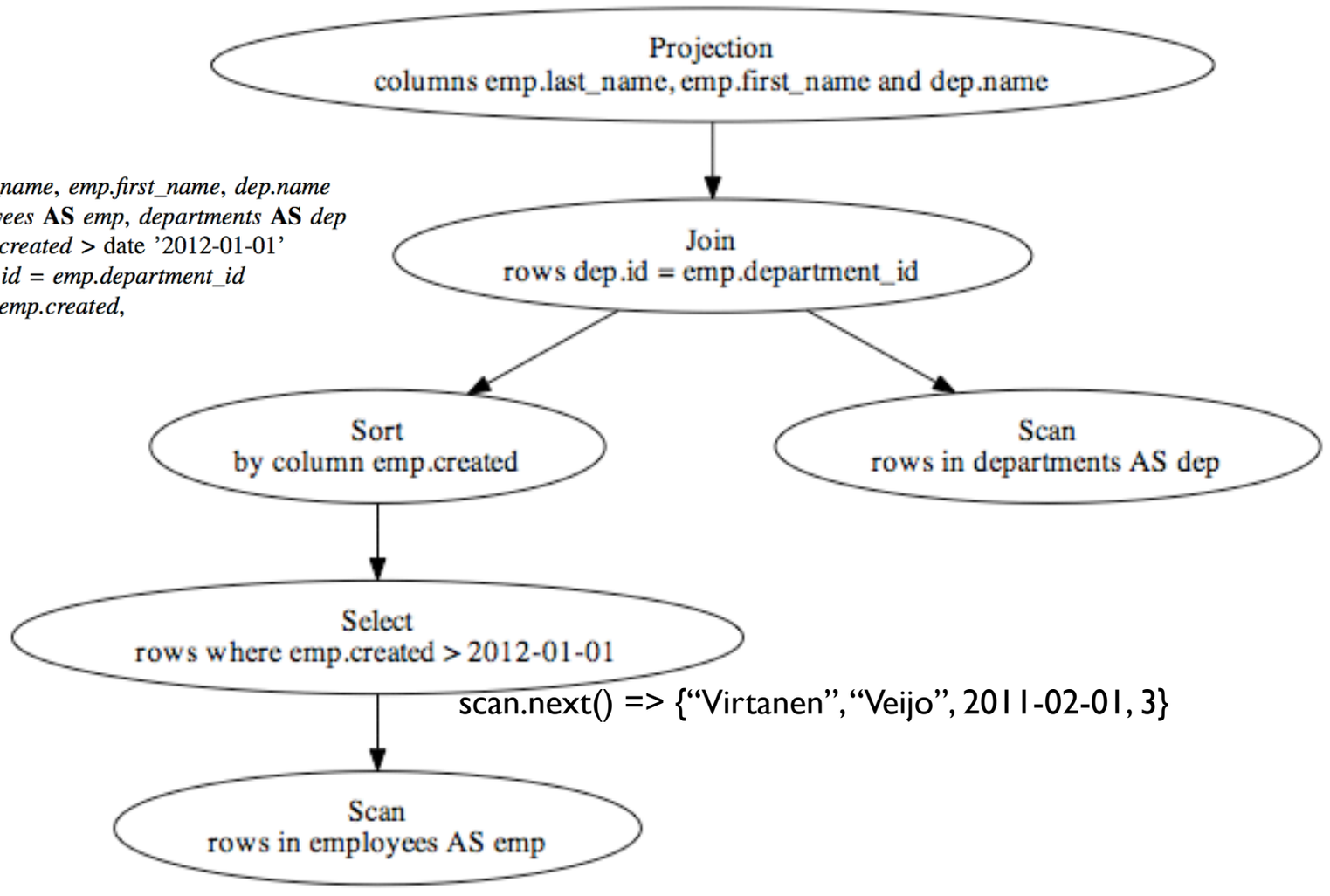
Query operator tree

```
SELECT emp.last_name, emp.first_name, dep.name
    FROM employees AS emp, departments AS dep
    WHERE emp.created > date '2012-01-01'
        AND dep.id = emp.department_id
     ORDER BY emp.created,
```



Query operator tree

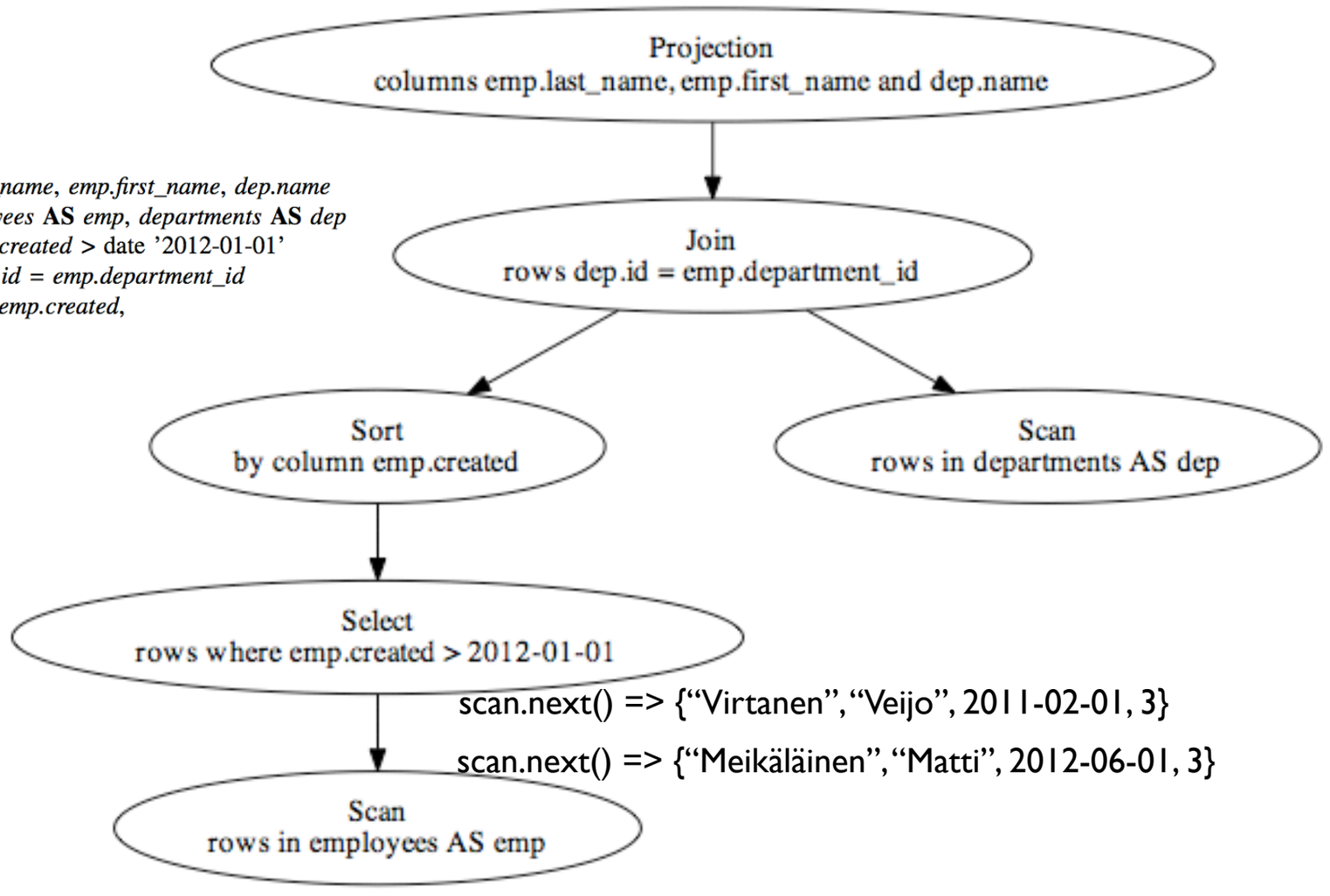
SELECT *emp.last_name*, *emp.first_name*, *dep.name*
FROM *employees* **AS** *emp*, *departments* **AS** *dep*
WHERE *emp.created* > date ’2012-01-01’
AND *dep.id* = *emp.department_id*
ORDER BY *emp.created*,



scan.next() => {“Virtanen”, “Veijo”, 2011-02-01, 3}

Query operator tree

SELECT *emp.last_name*, *emp.first_name*, *dep.name*
FROM *employees* **AS** *emp*, *departments* **AS** *dep*
WHERE *emp.created* > date '2012-01-01'
AND *dep.id* = *emp.department_id*
ORDER BY *emp.created*,

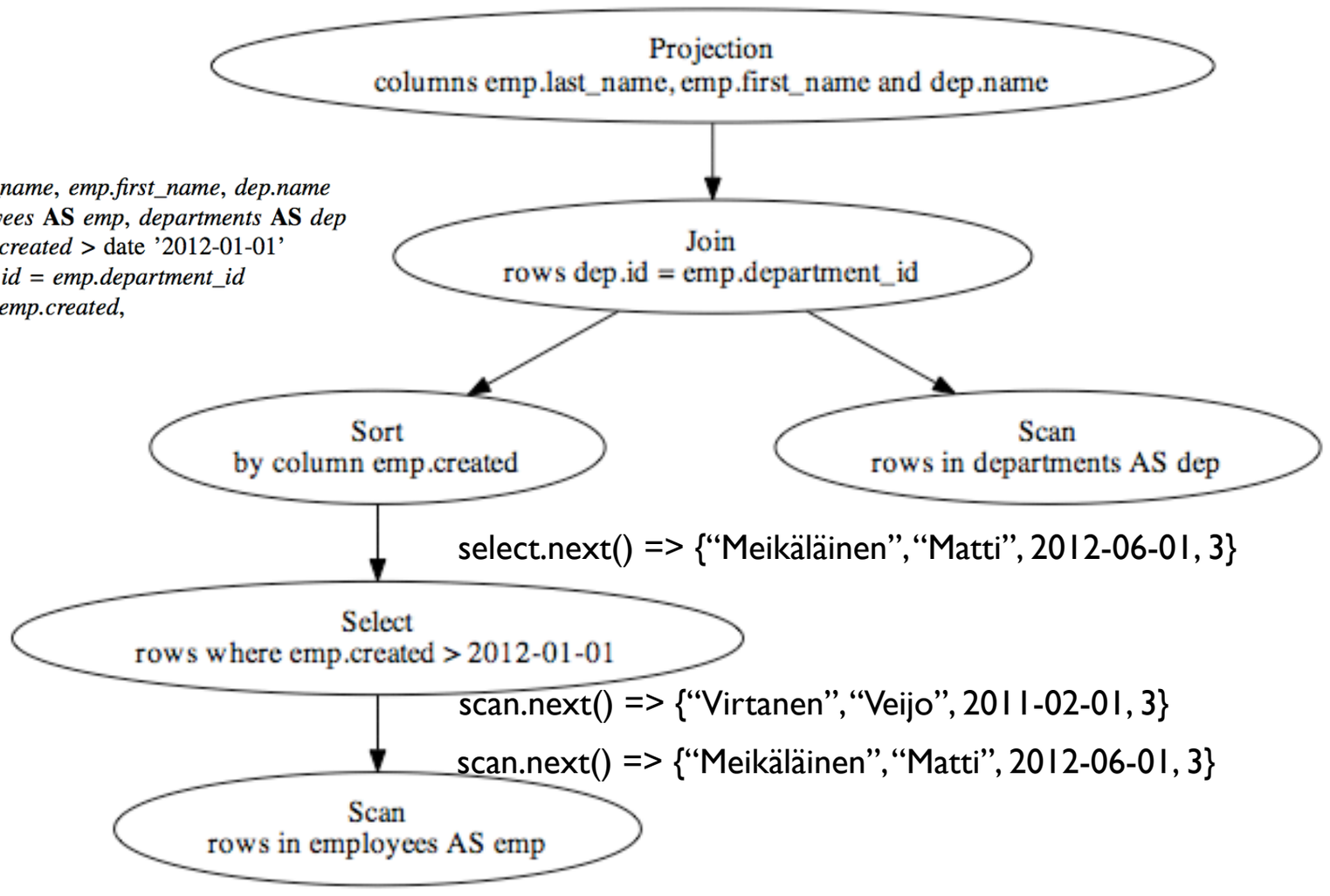


scan.next() => {"Virtanen", "Veijo", 2011-02-01, 3}

scan.next() => {"Meikäläinen", "Matti", 2012-06-01, 3}

Query operator tree

```
SELECT emp.last_name, emp.first_name, dep.name
    FROM employees AS emp, departments AS dep
    WHERE emp.created > date '2012-01-01'
        AND dep.id = emp.department_id
     ORDER BY emp.created,
```



select.next() => {"Meikäläinen", "Matti", 2012-06-01, 3}

scan.next() => {"Virtanen", "Veijo", 2011-02-01, 3}

scan.next() => {"Meikäläinen", "Matti", 2012-06-01, 3}

Volcano model

- Each query operator provides an iterator interface
- Iterator returns tuples from the disk
- Conceptually simple, beautiful
- Optimizations focused on the query plan level: avoid full table scans, minimize the amount of tuples processed

Problems with Volcano

- A query heavy of aggregation operators (AVG, SUM, ...) may spend only 10% of time averaging and summing, i.e. doing actual work

- MIPS R12000 can do a double multiplication in 3 cycles, MySQL takes 49 to do that; no loop pipelining!

cum.	excl.	calls	ins.	IPC	function
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Table 2: MySQL gprof trace of TPC-H Q1: `+,-,*,SUM,AVG` takes <10%, low IPC of 0.7

Column-oriented processing

- Predicates in Scan operators
- Late tuple materialization
- Invisible joins
- Operations on compressed data

Predicates on Scan operators

- Possible to do exact matches on heavily-compressed data (LZ-encoding)
- Can avoid dictionary lookups in a similar fashion
- Operating on run-length or bit-vector - encoded columns is possible when the predicate matcher knows about the compression used

Late tuple materialization

- Operators operate on position lists
- Join position lists and materialize tuples at the very end
- Position lists are trivial to produce from sorted columns (<, >, ==)
- Position lists can be coded as bitmaps for CPU-efficiency

Invisible join

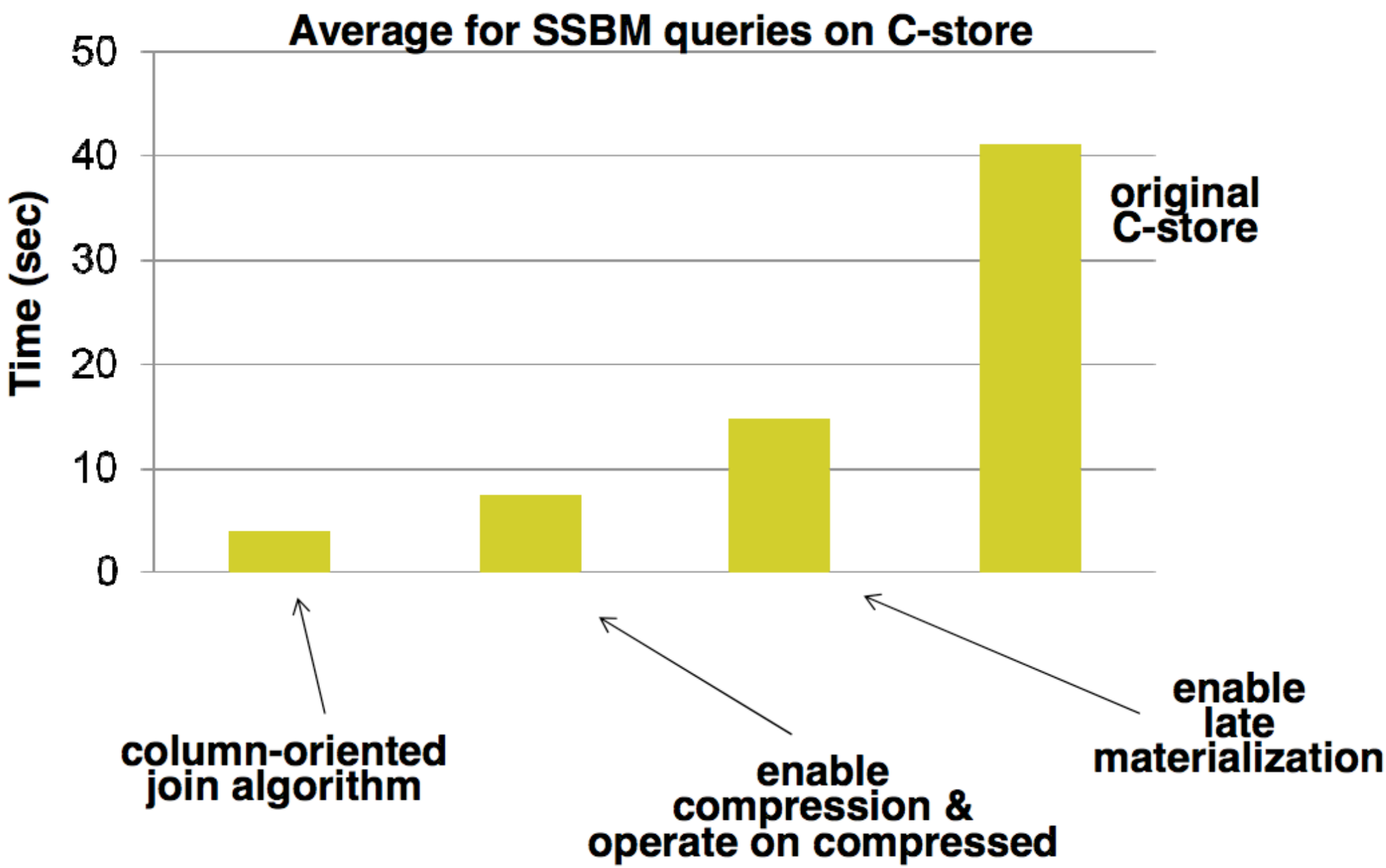
- Compute a bitmap (position list) for select predicates
- Join result is the intersection of bitmaps
- Results can be calculated efficiently by bitmap operations
- Useful in column-stores and data warehouses where joins of facts & dimensions

Operating on compressed data

- Push predicate down to Scan operator
- Don't decompress when not needed
  - Dictionary encoding only needs to decompress once
- Keep a cache of decompressed values
- Makes it possible to store columns in multiple sort orders

Performance benefits of...

- Late materialization
- Compression
- Invisible join


Average for SSBM queries on C-store
Time (sec)
50
40
30
20
10
0
original C-store
column-oriented join algorithm
enable compression & operate on compressed
enable late materialization

Alternative design

- C-Store
  - Column-optimized Query operators
  - Late tuple materialization
  - Modified Scan operators
- MonetDB/X100
  - Query execution as array manipulation
  - Emphasis on vectorized processing and high CPU efficiency

MonetDB/X100, solving memory bottleneck

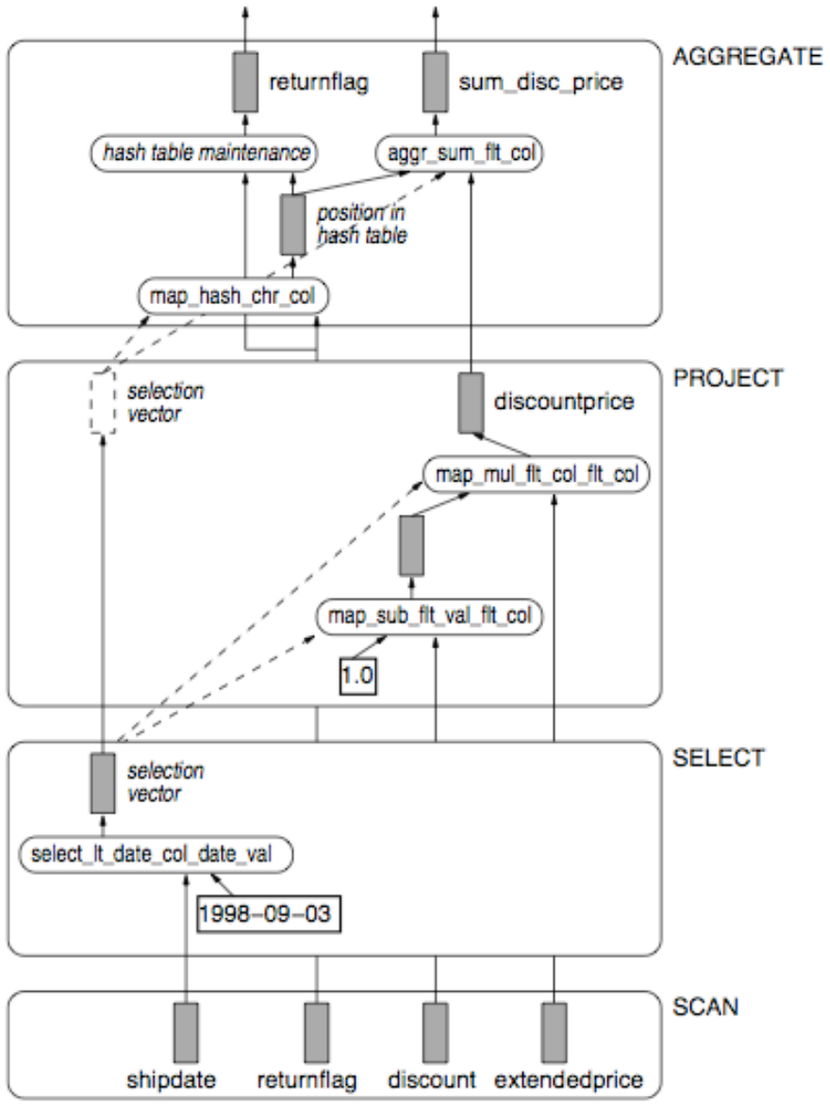
- Operators work with chunks of data that fit in the CPU cache (~1024 values)
- Operators are vectorized, have low degree of freedom (are simple, don't handle arbitrary predicates etc) in order for the compiler to be able to do loop pipelining
- Decompress pages to CPU cache, not RAM

X100 query tree

- Still a pull-model, but based on vectors, not tuples or values

- Emphasis on CPU cache efficiency

- Enables Single-Instruction-Multiple-Data (SIMD) instructions



Recap

- Operator changes
  - Scan operator that knows of compression and can handle predicates
- Late tuple materialization
- Invisible joins
- Operations on compressed data
- Vectorized processing