The evolution of vertical database architectures – a historical review

Per Svensson

Swedish Defence Research Agency

FOI

Outline

- Modern RDBMS design rules
- Transposed files
- The Decomposed Storage Model (DSM)
- The impact of modern processor architecture
- MonetDB's radix cluster algorithm for hash join
- Data compression
- Compression techniques
- Compressed, fully transposed, ordered files (CFTOF)
- Querying CFTOF data
- Experiment 1: Cantor vs. Mimer (1988)
- Experiment 2: SkyServer performance using MonetDB (2007)
- Vectorization in MonetDB/X100
- The vectorized data-flow execution model
- Conclusion

Reference

- Per Svensson, Peter Boncz, Milena Ivanova, Martin Kersten, Niels Nes**:** *Emerging “vertical” database systems in support of scientific data*
- in Arie Shoshani & Doron Rotem (eds.): *Scientific Data Management: Challenges, Existing Technology, and Deployment.* Taylor & Francis, in press.

FOI

Modern RDBMS design rules

Transactional databases:

- Store data row-wise to update records by single write operations whenever possible
- Small disk pages to minimize data transferred, as well as size of locked part of the disk
- Index few attributes to avoid locking entire index structures
- Not profitable to compress data due to mix of data types in each row
- Adding or deleting attributes and indexes likely to be expensive
- Attribute updates likely to be costly since entire row must be read and rewritten

"Complex analytics" databases:

- Use of column-wise storage makes it possible to avoid touching pages not affected by query
- Commonly accessed columns tend to stay in the cache
- Process attributes sequentially, store data in large pages => many relevant items retrieved in single read => higher overall hit ratio
- Few concurrent users => version management can replace record-wise locking techniques => queries see consistent database => very few locks
- Possible but seldom necessary to index every attribute
- CPU time "investment" in storage structures likely to be profitable
- Data compression profitable also because data are likely to be homogeneous
- Adding, deleting or updating attributes likely to be relatively cheap

Adapted from (MacNicol et al., VLDB04)

FOI

Transposed files

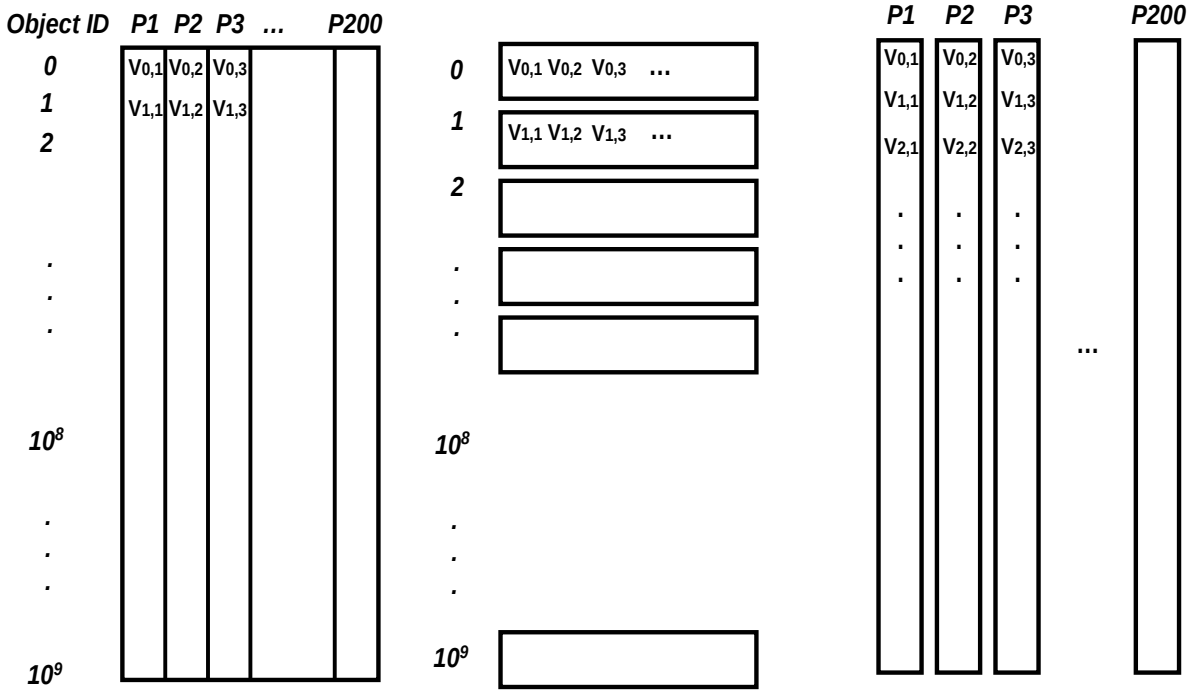


Fig. 11.1(a): large table

Fig. 11.1(b): horizontal layout

Fig. 11.1(a): vertical layout

Transposed files were popular in the ’70s for use in SSDBs (RM, TOD, RAPID, ALDS,…). The first relational system using transposed files may have been Cantor.

The Decomposed Storage Model (DSM)

- Introduced by Copeland & Khoshafian 1985
- A DSM is a *"[fully] transposed storage model with surrogates [TIDs] included"*
- Each column of a relational table stored in separate *binary association table* (BAT), as array of two-field records (TID, value)
- Two copies of each BAT are stored, one clustered / sorted on each of the attributes (TID, value)
- Used in MonetDB, C-Store and other systems

The impact of modern processor architectures

- Optimal use of cache memory becomes ever more critical as CPU speed increases outpace DRAM latency improvements
- It is no longer appropriate to think of the main memory as “random access” memory
- DBMSs have become compute and memory bound due to sophisticated techniques used for hiding I/O latency
- MonetDB’s *radix cluster algorithm for hash join* is a good example of ”cache-conscious” DBMS algorithms

FOI

Partitioned hash-join in MonetDB

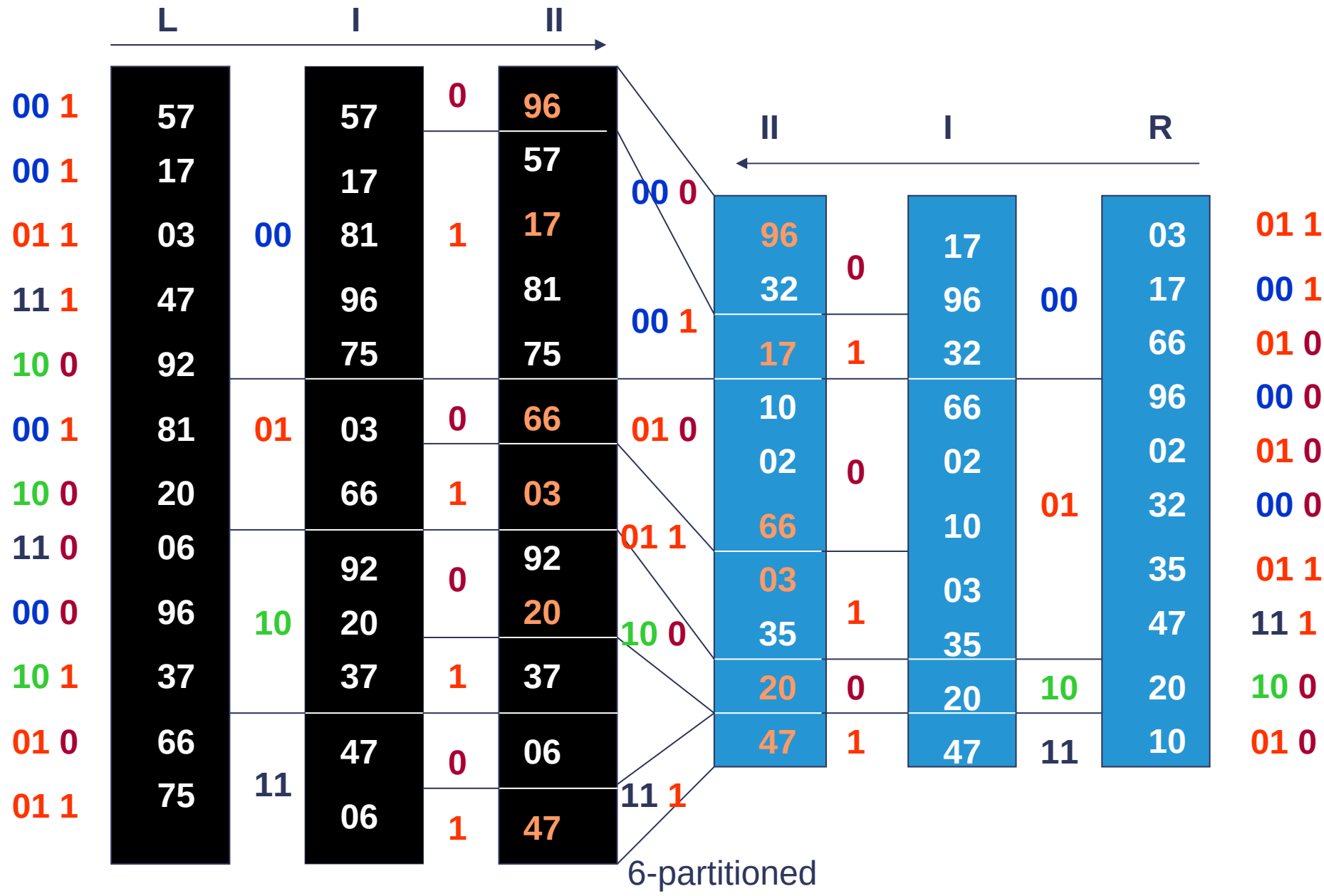
- Divide each operand relation into a sequence H of clusters using multiple passes
- Radix-cluster on lower B bits of the hash value. Requires P sequential passes, $\Sigma_1^P B_P = B$
- A radix-clustered relation is ordered on radix-bits
- Perform a merge step on the radix-bits of both radix-clustered relations to get the pairs to be hash-joined together

FOI

Partitioned hash-join in MonetDB II

```
partitioned-hashjoin(L, R, H)
    radix-cluster(L, H)
    radix-cluster(R, H)
    FOREACH cluster c IN [1..H]
        hash-join(L[c], R[c])
end
```

FOI


L
I
II
II
I
R
00 1
00 1
01 1
11 1
10 0
00 1
10 0
11 0
00 0
10 1
01 0
01 1
57
17
03
47
92
81
20
06
96
37
66
75
00
01
10
11
57
17
81
96
75
03
66
92
20
37
47
06
0
1
0
1
0
1
0
1
96
57
17
81
75
66
03
92
20
37
06
47
00 0
00 1
01 0
01 1
10 0
11 1
96
32
17
10
02
66
03
35
20
47
0
1
0
1
0
1
17
96
32
66
02
10
03
35
20
47
00
01
10
11
03
17
66
96
02
32
35
47
20
10
01 1
00 1
01 0
00 0
01 0
00 0
01 1
11 1
10 0
01 0
6-partitioned
2-pass radix-cluster
hash-join
2-pass radix-cluster

Data compression

- Compression improves I/O performance by:
  1. reducing disc seek times
  2. reducing I/O transfer times
  3. increasing buffer hit rate
- CPU overhead of (de)compression compensated by reduced I/O
- Storing columns in multiple sort orders may improve query performance

Compression techniques

- *Dictionary schemes* code literal attribute values in fewer bits
- *Huffman encoding* may be used for text, requires varying length codes
- *Frame of reference* encoding (*FOR*): values in a block of data stored as differences from a “frame of reference” value
- *Run length encoding* (RLE): repeats of the same element stored as (value, run-length) pairs – best effect together with sorting

Compression techniques II

In Cantor's testbed (1979), four alternatives were repeatedly evaluated during sequence write operations:

1) use minimum common byte length for subsequence, store byte length in header (FOR, "bit packing")
2) use minimum common byte length for *difference* subsequence, store first element and byte length in header (FOR-delta)
3) if subsequence is *run* of equal values, store value and length in header (RLE)
4) if subsequence is run of equal differences, store first element, difference, and length in header (delta-RLE)

FOI

Compression techniques III

- A *dynamic programming, branch-and-bound* algorithm determined how to store a given integer sequence of length n in minimal space given the constraints
- The sequence was subdivided into subsequences, each characterized by *storage alternative*, *byte size*, *cardinality*, and *size of header*
- The problem was solved in time O(n)

Compressed, fully transposed, ordered files

- Batory (VLDB 1978, TODS 1979) showed that search algorithms for *transposed files* could outperform index-based ones in a large proportion of cases
- Svensson (VLDB 1979) showed that conjunctive queries may be evaluated even faster by use of a *compressed, fully transposed ordered file* (CFTOF) storage structure
- The performance of a test-bed was compared with a commercial database system and an analytical model
- Results showed that order-of-magnitude performance gains could be achieved

Querying CFTOF data

Let the relation R have three key attributes, v1, v2, v3.

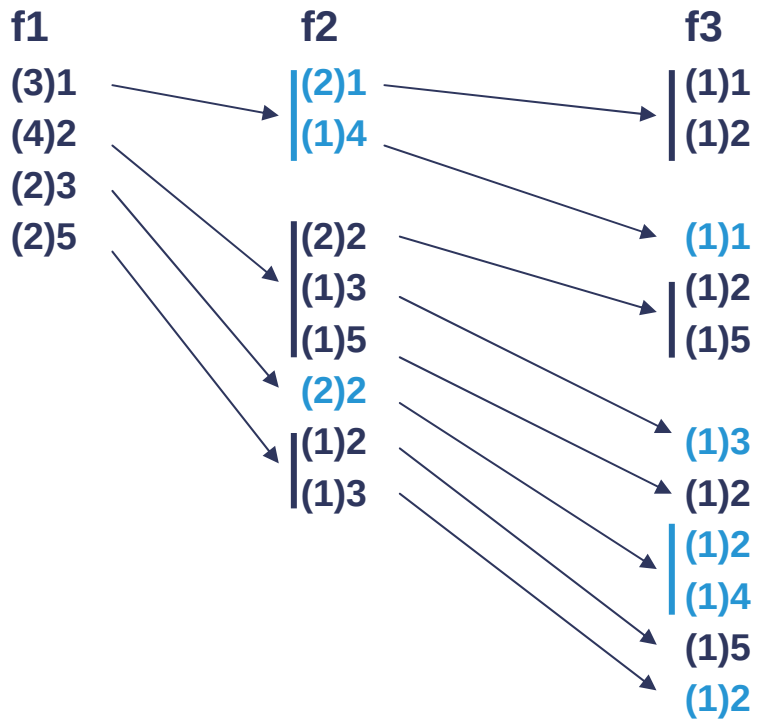
After sorting we may have:

v1	v2	v3	TID
1	1	1	1
1	1	2	2
1	4	1	3
2	2	2	4
2	2	5	5
2	3	3	6
2	5	2	7
3	2	2	8
3	2	4	9
5	2	5	10
5	3	2	11

FOI

Querying CFTOF data II

RLE-compressing each column gives (respecting hierarchy):



Querying ordered, run-length compressed data III

Assume we have the conjunctive search predicate:

(2 <= v1 <= 3) & (v2 >= 3) & (v3 >= 2)

p1 & p2 & p3

1. Scan f1 w r t p1, leaves ((4)2, (2)3) ->T4-9
2. Scan f2 | T4-9 w r t p2, leaves ((1)3, (1)5) -> T6-7
3. Scan f3 | T6-7 w r t p3, leaves ((1)3, (1)2) -> T6-7

FOI

CFTOF search performance

Seven random files of different size were drawn from $[1..10]^4$

Range queries with different selectivity were evaluated against each file

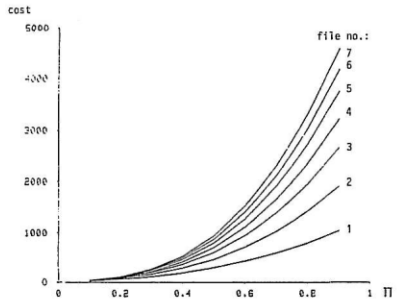


Fig. 3.1. Theoretical cost as function of permissivity for the seven test files in Example 3.1.

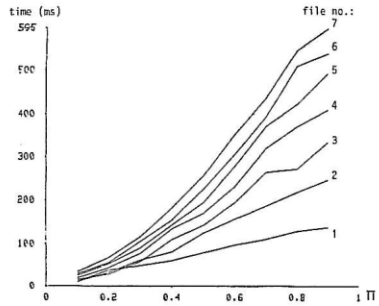


Fig. 3.2. Measured search times a. function of permissivity for the seven test files.

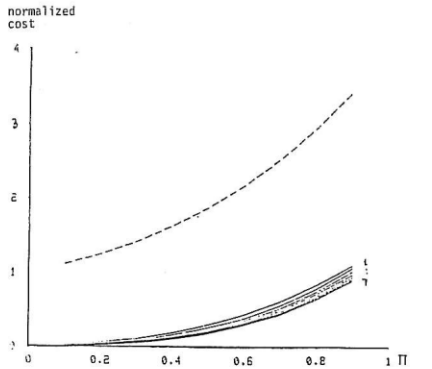


Fig. 3.3. Normalized cost as function of permissivity for the seven test files in Example 3.1 (unbroken lines) and for uncompressed transposed file search (dashed line).

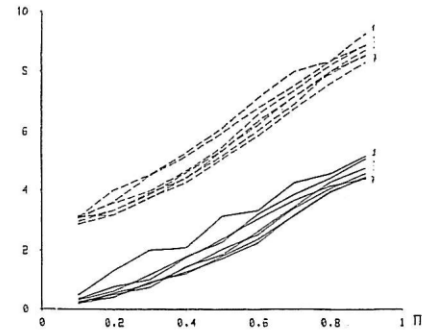


Fig. 3.4. Normalized measured search times for compressed (unbroken) and uncompressed (dashed) ordered transposed files.

FOI

Experiment 1: Mimer *vs.* Cantor (SSDBM 1988)

Query % hits	Q1 0.1 %	Q2 1.0 %	Q3 10 %	Q4 50 %
File size 12 ktup	0.10	0.51	3.2	7.9
File size 20 ktup	0.16	0.82	3.6	8.8

Table of CPU-time$_{Mimer}$ / CPU-time$_{Cantor}$

FOI

Experiment 2: SkyServer benchmark using MonetDB (SSDBM 2007)

	Table scan 1.5GB	Index scan 1.5GB	Table scan 150GB	Index scan 150GB
Row-store	6.6	0.4	245	24
Column-store	0.4	0.47	53	16

FOI

Vectorization in MonetDB/X100

- Use of vector operators (vectorization) in query evaluation aims at distributing interpretation overhead over many CPU operations
- A vectorized *prototype* query processor called X100 was recently built and evaluated by the MonetDB developers
- The goals of MonetDB/X100 are to:
  1. ***execute high-volume queries at high CPU efficiency***
  2. ***be extensible to application domains like data mining and multi-media retrieval***
  3. ***scale with the size of the lowest storage hierarchy (disc)***

Vectorization II

To achieve these goals, MonetDB/X100 manages bottlenecks throughout the architecture:

- ***Disc***. The I/O subsystem is oriented towards efficient sequential data access
- Uses vertical storage layout sometimes enhanced with lightweight data compression
- ***RAM***. RAM access carried out by explicit memory-to-cache routines with platform-specific optimizations
- The same data layout as used on disc is employed in RAM to save space and bandwidth

FOI

Vectorization III

- ***Cache***. Vertical chunks (*e.g.*, 1000 values), “vectors”, are units of operation
- In CPU, cache bandwidth does not matter, therefore (de)compression occurs on boundary between RAM and cache
- ***CPU***. Vectorized primitives show compiler that processing a tuple is independent of previous and subsequent tuples
- The X100 design strives to achieve vectorization also for other operators than projections (*e.g*., aggregation)

FOI

Data-flow execution model

- Karasalo and Svensson (SSDBM 1986) survey methods used in Cantor for ***network generation,*** *i.e.,* translating the syntax tree of a *parsed query* into an execution plan
- The execution plan consists of one or more hierarchies of *vectorized dataflow networks*
- Network generation is followed by an execution phase which proceeds in two stages, *buffer allocation* and *network evaluation*

Data-flow execution model II

- Initially all but upstream *boundary buffer nodes* are empty
- ***Network evaluation*** executes operator nodes in some order until all downstream boundary buffer nodes contain a value
- An operator node may execute whenever none of its inbuffer nodes is empty, and none of its outbuffer nodes is full

FOI

Data-flow execution example

Given data:

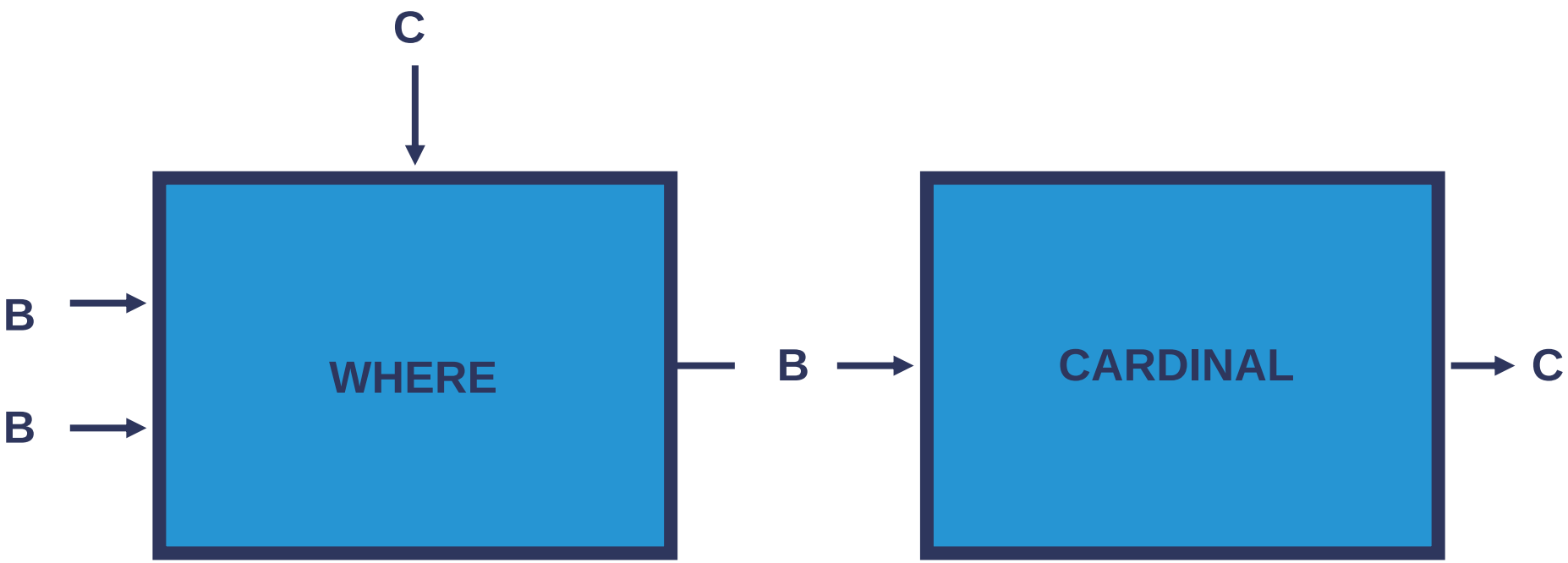
- *BASERELATION Sales ((dept:LITERAL, item: INTEGER), vol: INTEGER);*
- *BASERELATION Location((dept:LITERAL), floor: INTEGER);*

Compute the set of items sold by all depts on 2nd floor!

- *VIEW ndep2 <-CARDINAL(Location WHERE [floor=2]);*
- *VIEW dep2itms <- *(s:Sales, l:Location) WHERE [(s.dept=l.dept) AND (l.floor=2)] .[dept:s.dept, item:s.item];*
- *VIEW ndepitm <- dep2itms BY [item:item] COMPUTE .[item:item, ndeps:CARDINAL];*
- *VIEW itmfl2 <- ndepitm WHERE [ndeps=ndep2][item];*

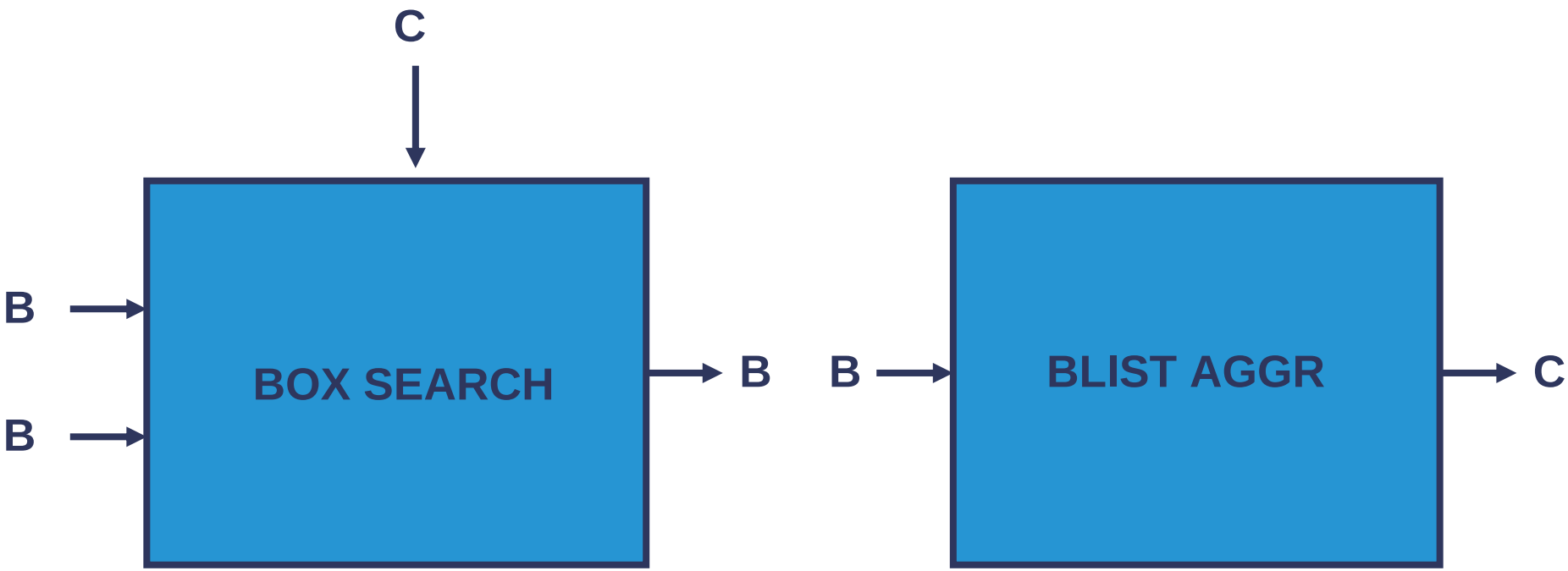
FOI

Data-flow execution example II

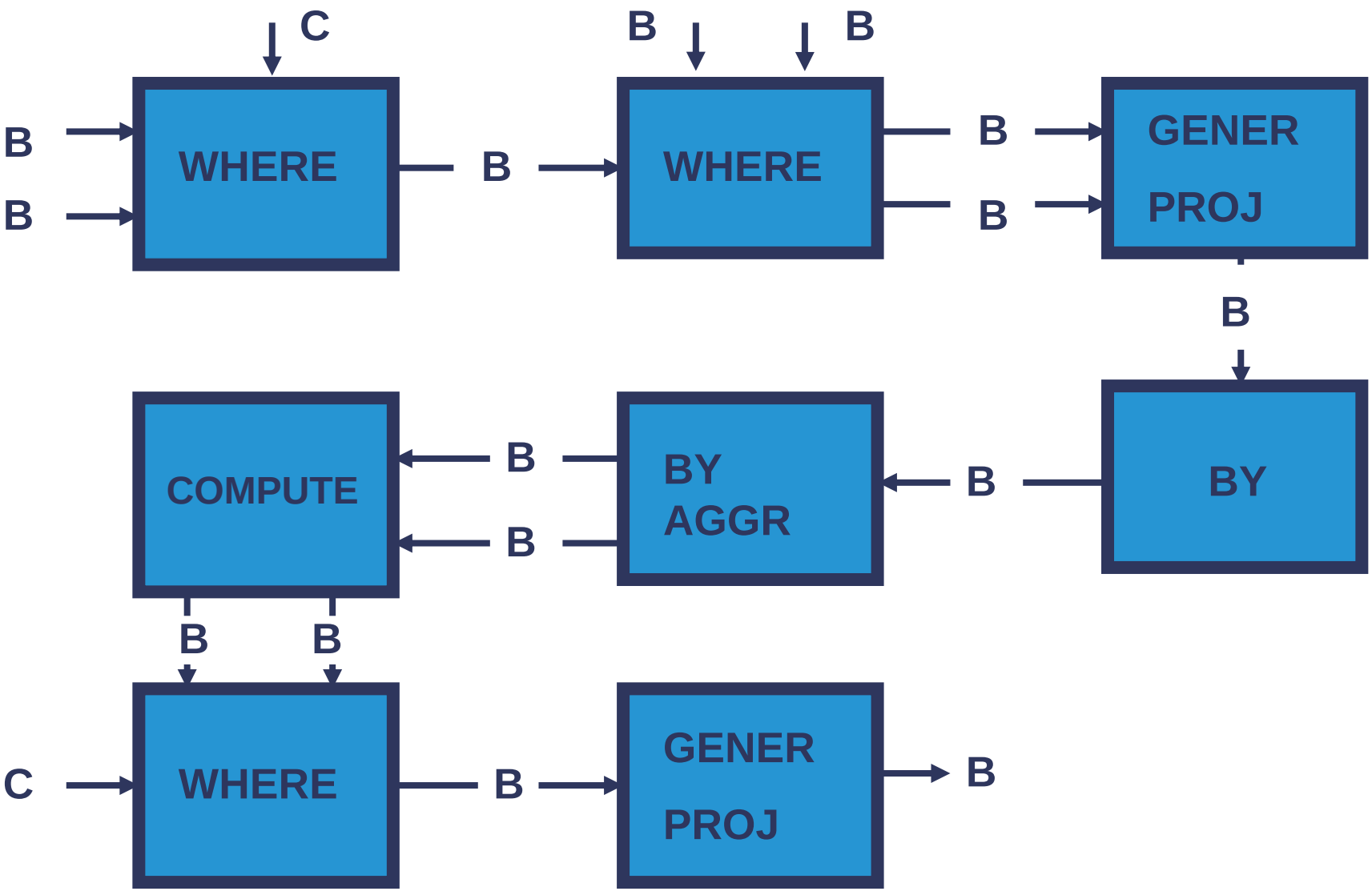


Root level net of first hierarchy generated for itmfl2

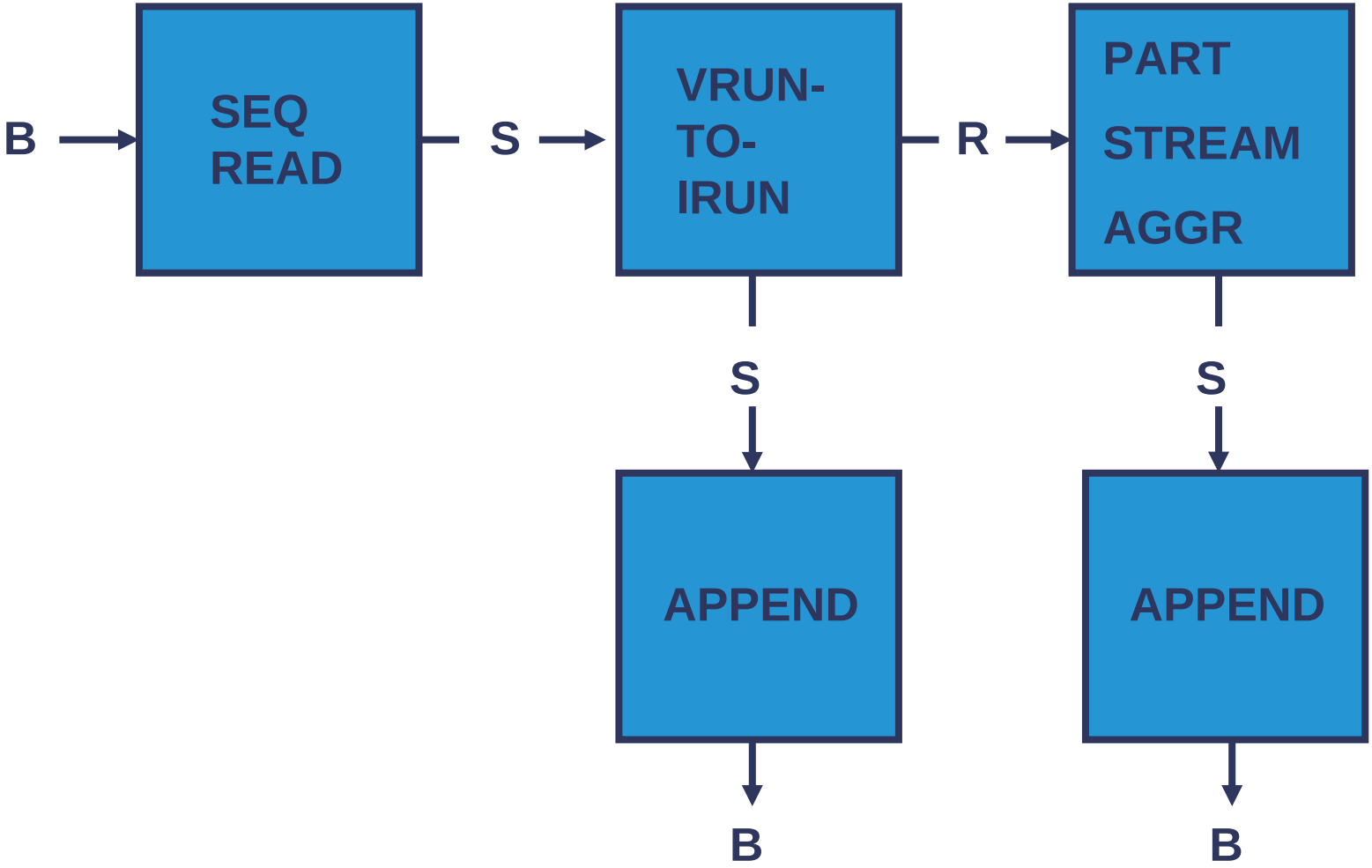
Data-flow execution example III



FOI



Root level net of second hierarchy generated for itmfl2



BY-AGGREGATE subnet

Conclusion

- Modified RDBMS design rules have recently been proposed for “complex analytics” applications (MacNicol et al., VLDB04)
- The Decomposed Storage Model (1985) has lately become a cornerstone of “vertical” relational database design
- *But* transposed files were used successfully already in the early ´70s
- Modern CPU architecture properties calls for unconventional, benchmark-supported design solutions and new query processing algorithms
- Data compression is becoming an important class of approaches in vertical RDBMS
- *But* was exploited, also in search and join algorithms, in CFTOF (1979)
- Vectorization and data-flow execution are important ways to further improve vertical RDBMS performance
- Benchmarks show greatly improved performance in data warehousing using vertical RDBMS solutions
- The ill-founded consensus around the superiority of the n-ary storage model is now, albeit belatedly, making room for a more balanced view
- Cantor did not include separate indexes because of clearly demonstrated CFTOF search advantages. Would that decision still stand?

FOI