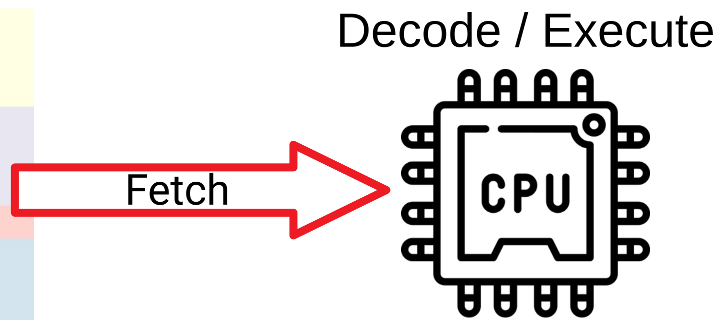
Database Administration

José Orlando Pereira

Departamento de Informática
Universidade do Minho

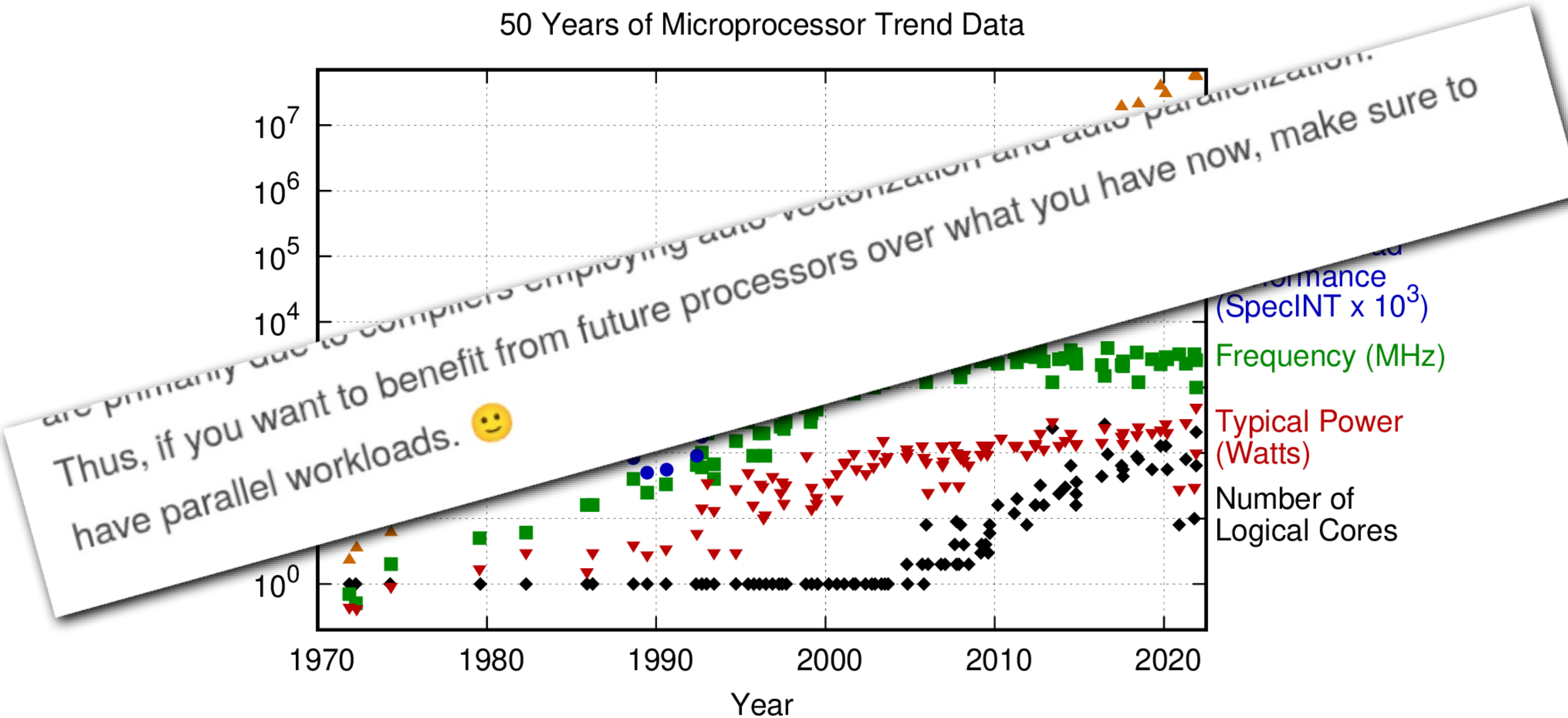
A model of computing

```
        mov     eax, DWORD PTR [rbp-20]
        imul    eax, eax
        mov     DWORD PTR [rbp-8], eax
        mov     eax, DWORD PTR [rbp-20]
        add     eax, eax
IP ->   mov     DWORD PTR [rbp-12], eax
        mov     DWORD PTR [rbp-4], 5
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, DWORD PTR [rbp-12]
        test    eax, eax
        jns     .L2
        mov     eax, DWORD PTR [rbp-12]
        add     DWORD PTR [rbp-4], eax
.L2:
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     edx, eax
        mov     eax, DWORD PTR [rbp-12]
```



- Challenge for data-intensive programs:
  - Computation is not fast enough

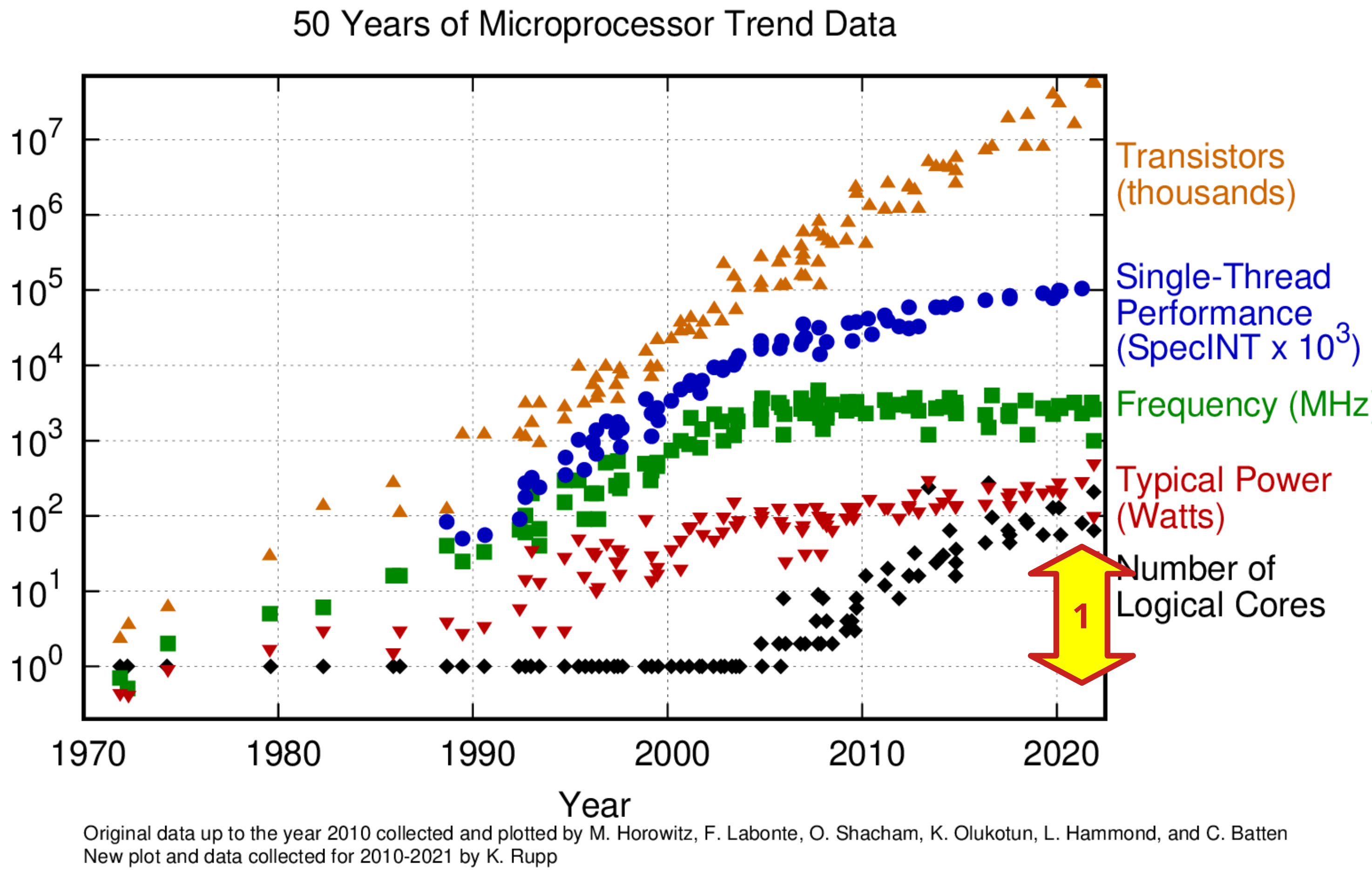
Moore's Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

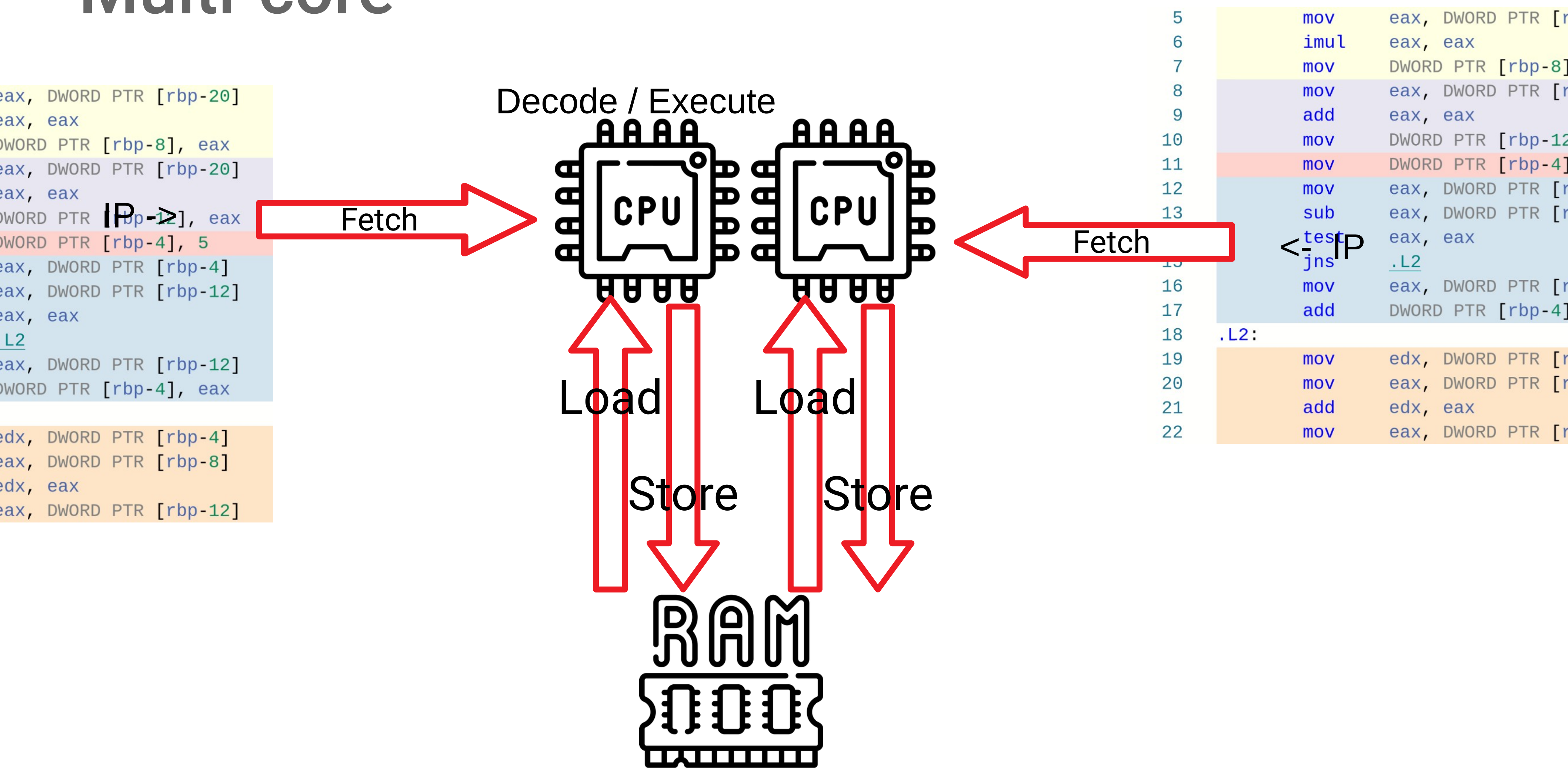
Source https://github.com/karlrupp/microprocessor-trend-data

Multiple cores



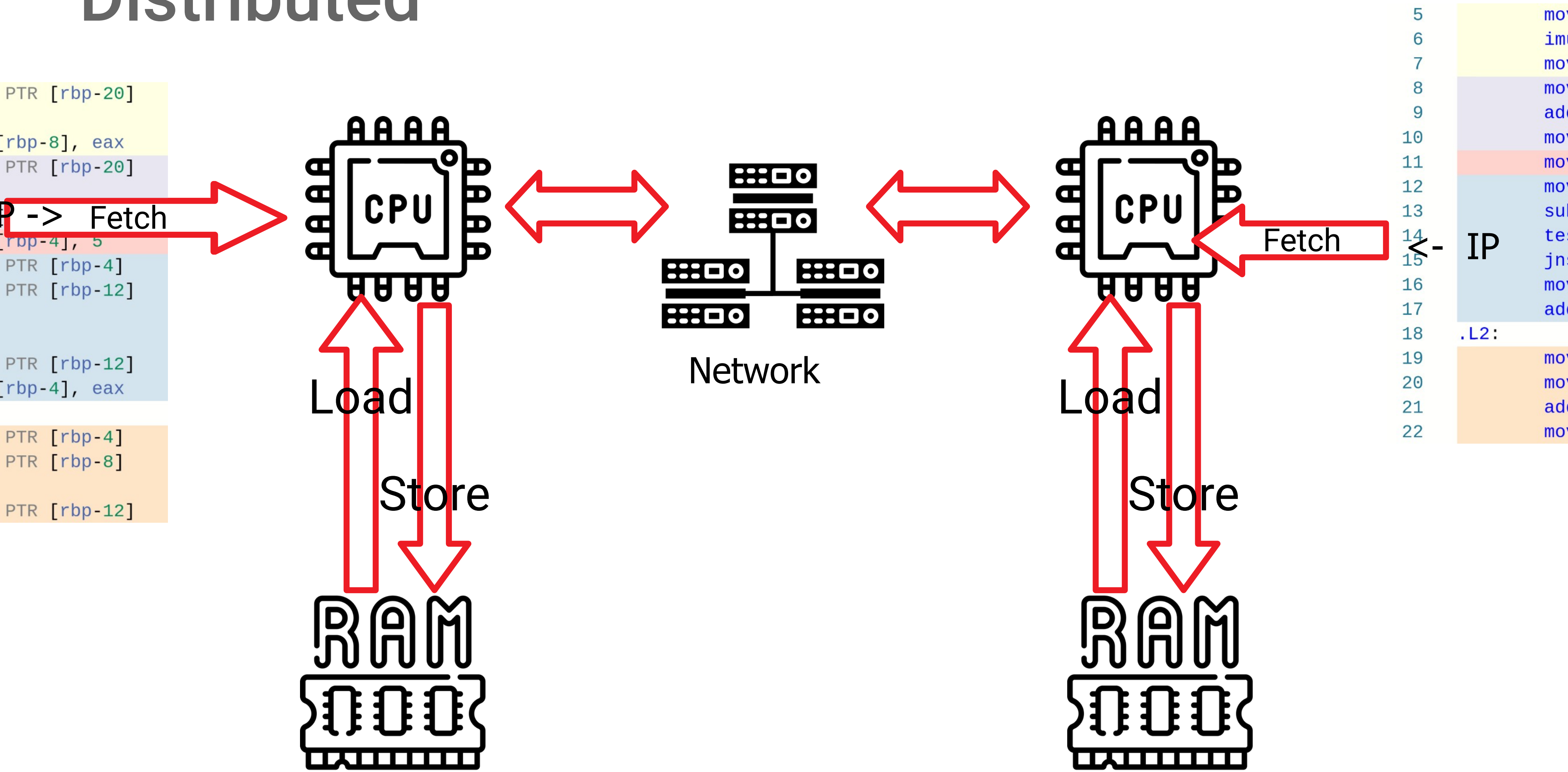
Source https://github.com/karlrupp/microprocessor-trend-data

Multi-core



Icons by Flaticon.com.

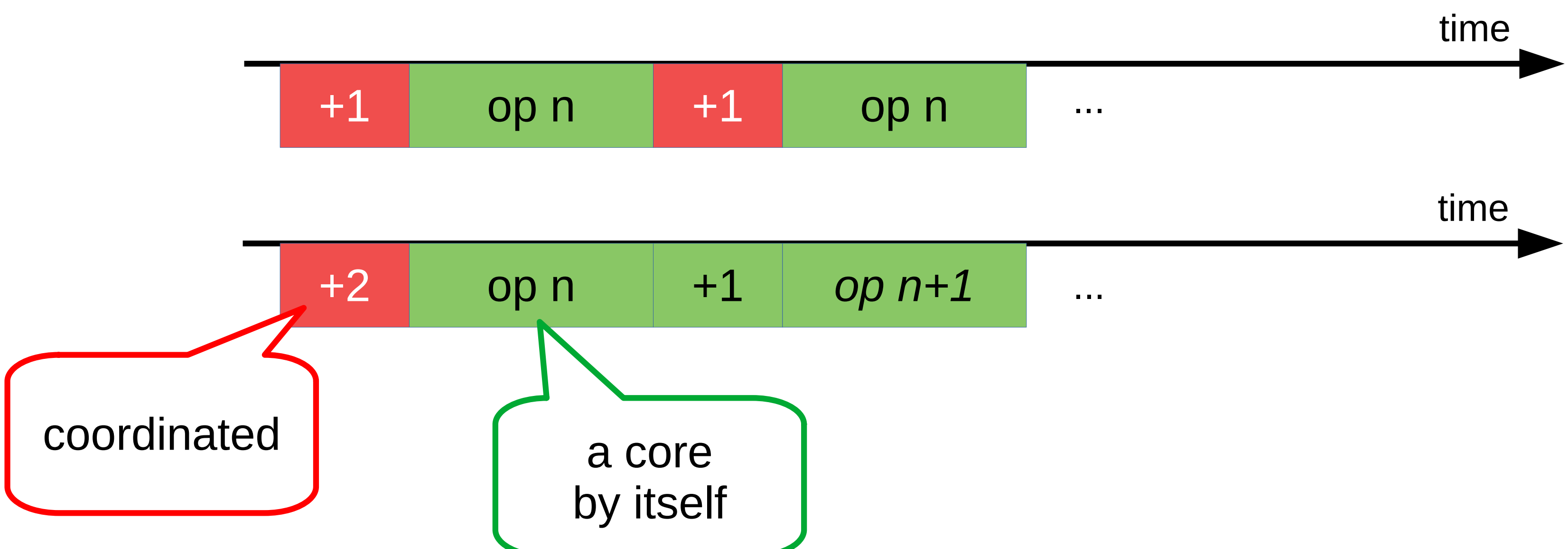
Distributed



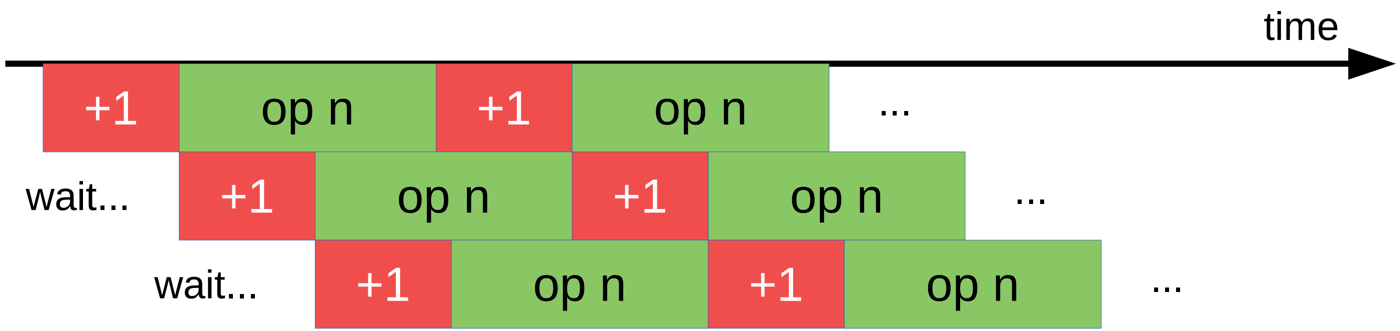
Icons by Flaticon.com.

Coordination overhead

- Splitting a task incurs in coordination overhead
- Consider two versions of a chunked vector operation:
  - Get chunk of size 1, execute
  - Get chunk of size 2, execute one and the other

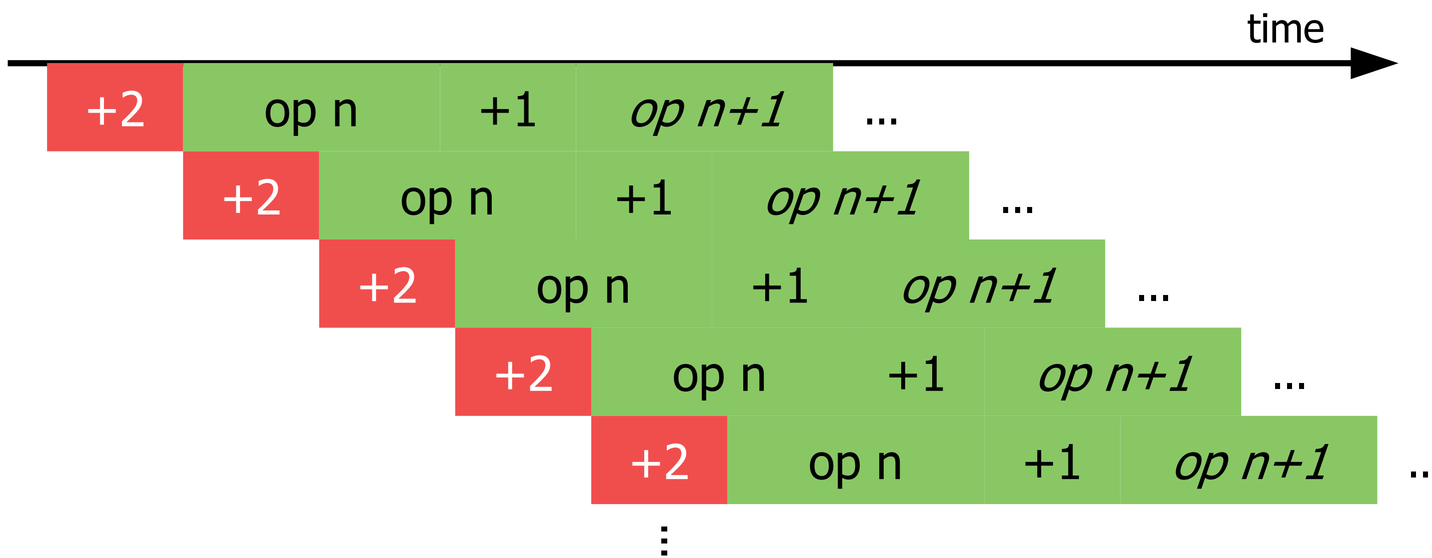


Coordination overhead



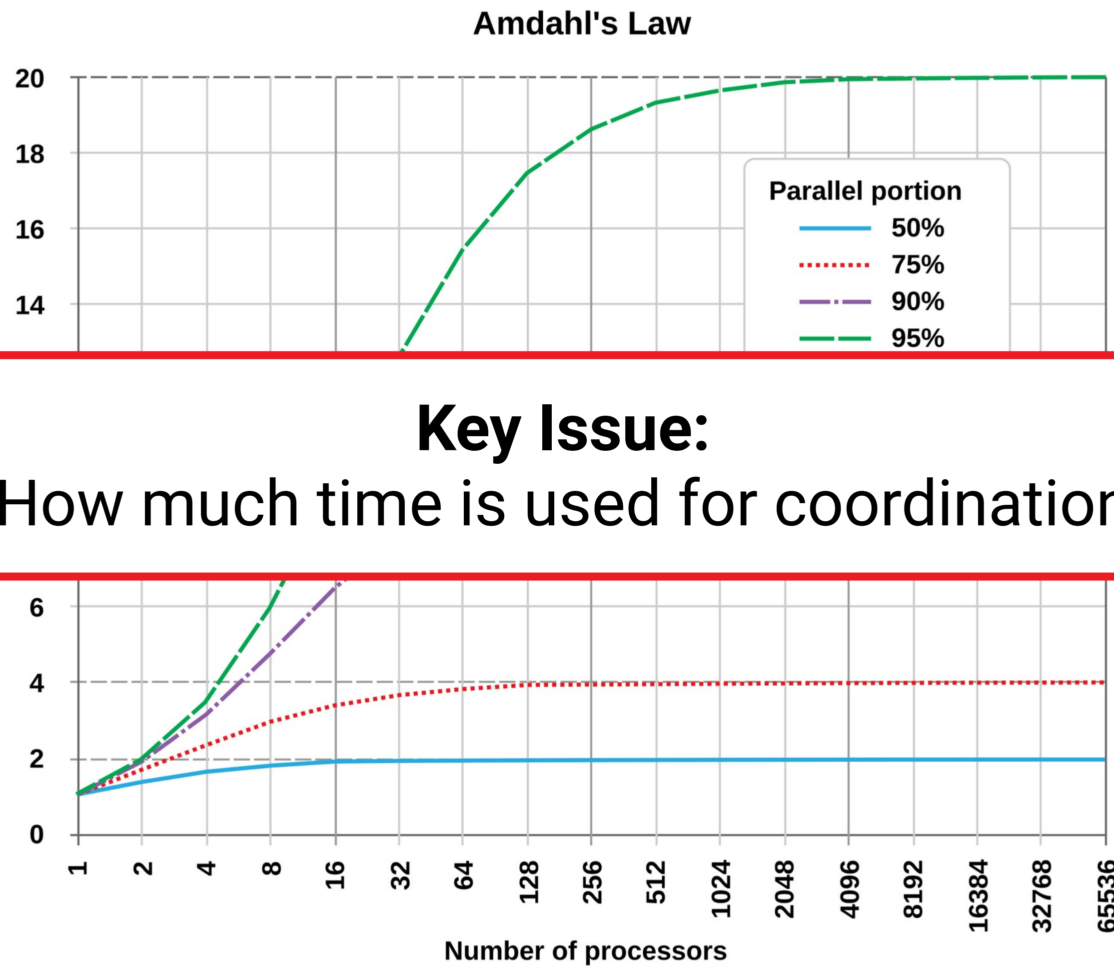
- Eventually, at least one core is blocked waiting for coordination

Coordination overhead



- Reducing the contention on coordination improves performance, even if doing the same work!

Amdahl’s Law



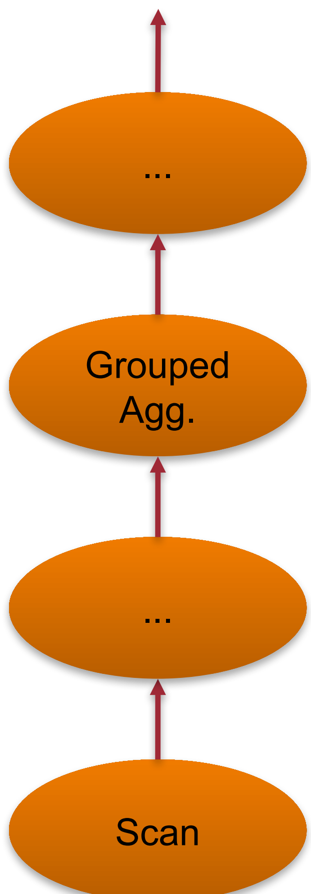
Key Issue:
How much time is used for coordination

Image from Wikipedia.

Parallel execution

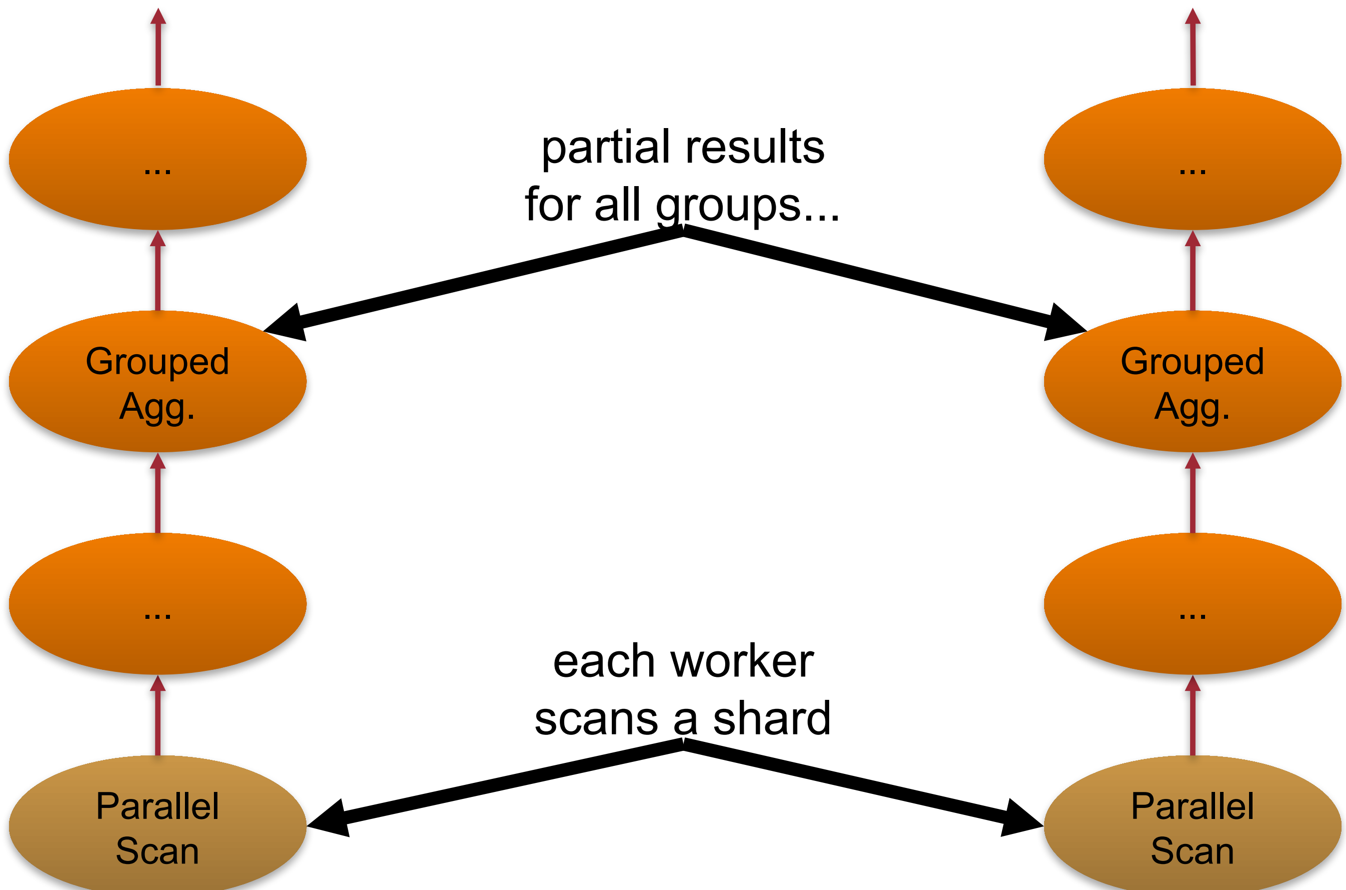
- Naive approach:
  - Split data in “shards”
  - Run an separate instance of the plan in each shard
  - Collect all results
- Sharding by:
  - Adding an additional filter to each scan when data is shared
  - Using the locally stored data when data is distributed

Example




```
select a, avg(b) from t
    where c > 0
    group by a
    having avg(b) > 0
```

Example: Naive parallel execution

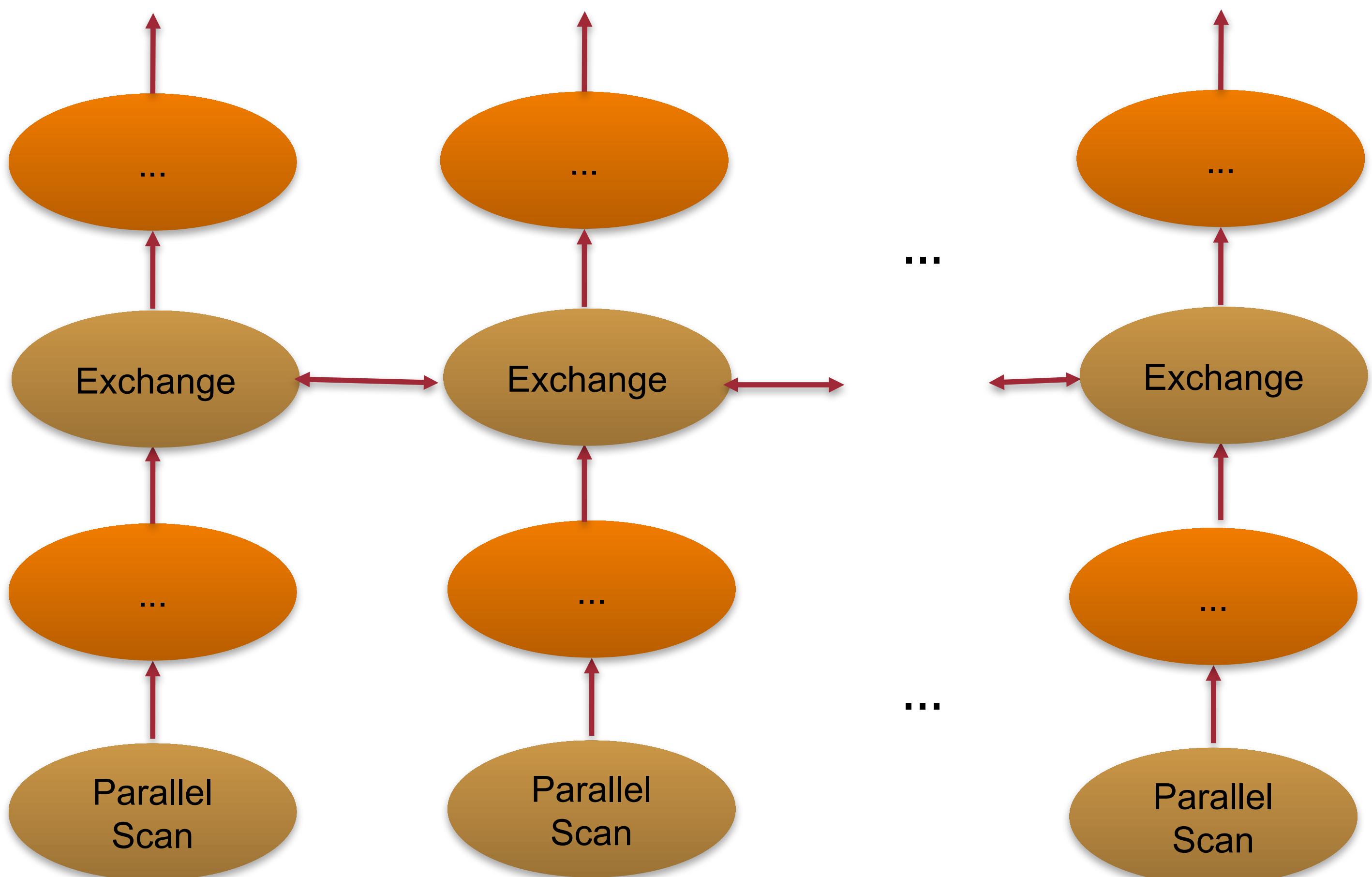


Volcano “Exchange” operator

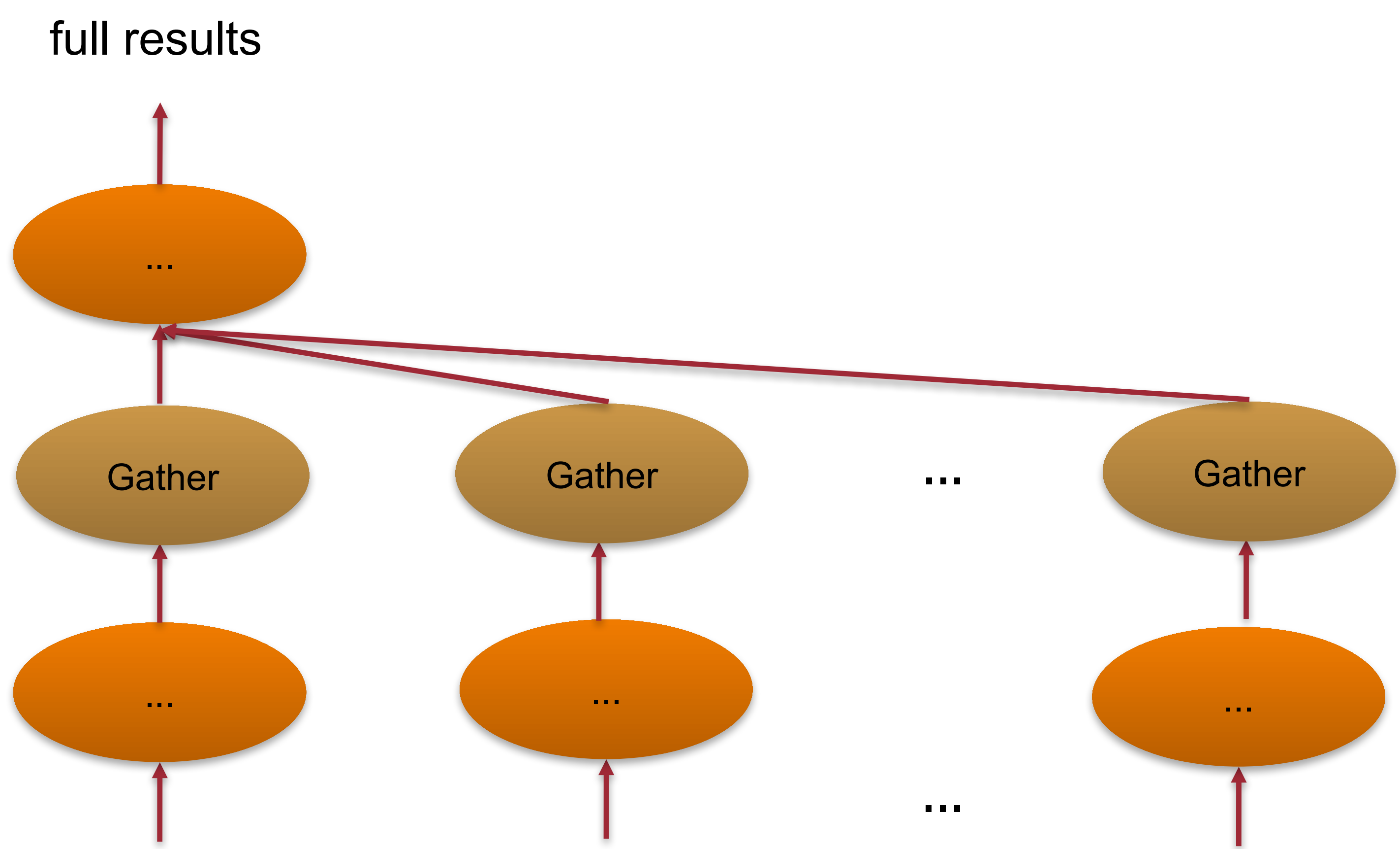
- Extends the iterator model to multi-core and distributed settings:
  - Contains an asynchronous, thread-safe (or distributed) buffer
  - Consumes data from multiple input iterators
    - Possibly each in a different copy of the plan
  - Can be consumed by multiple output iterators
    - Possibly each in a different copy of the plan
- Configurable rule to route tuples from inputs to outputs
- Different modes:
  - only one consumer → a.k.a. “gather”
  - respects order → a.k.a. “merge”
  - copies to all consumers → a.k.a. “broadcast”

Example: "Exchange" operator

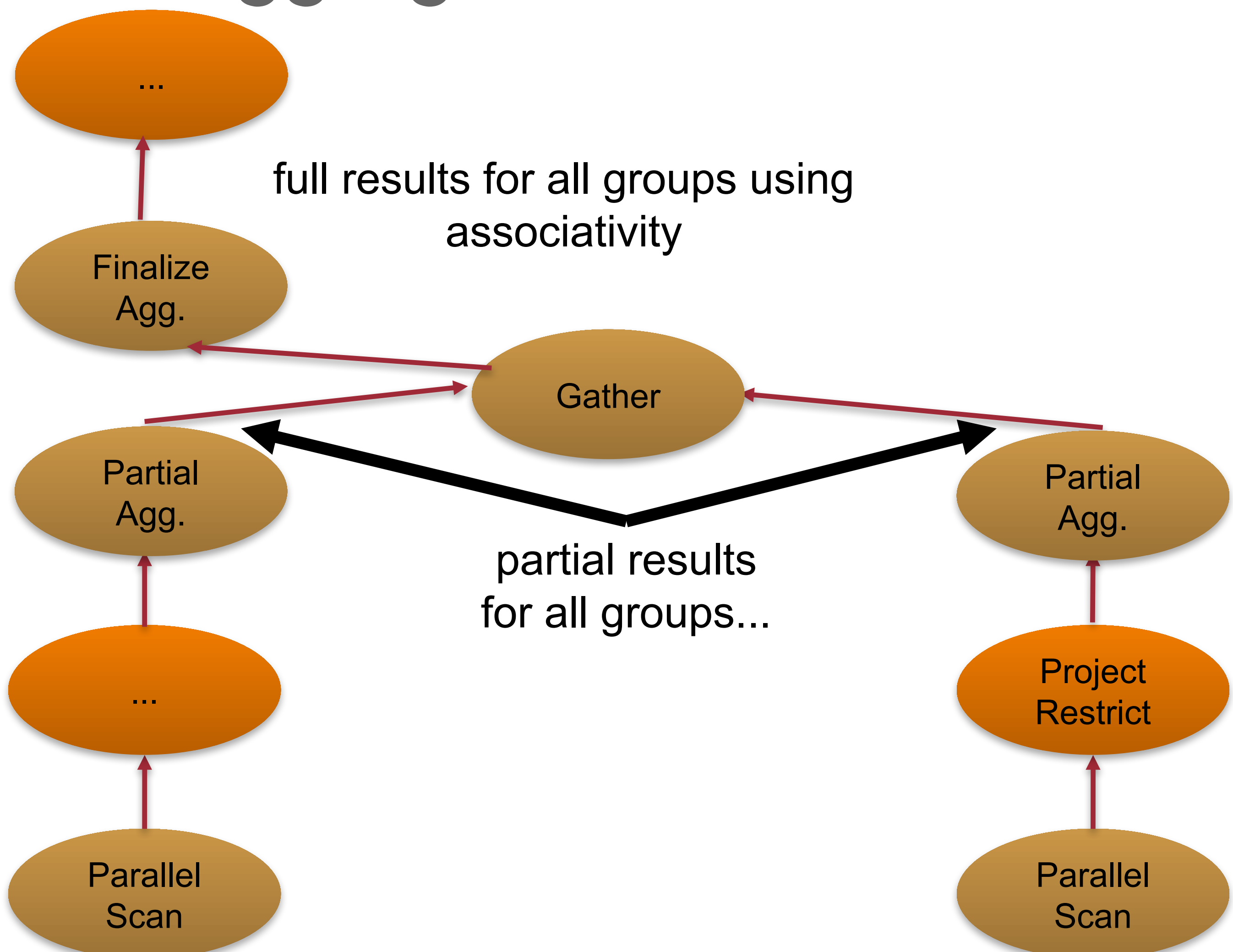
full results for some groups



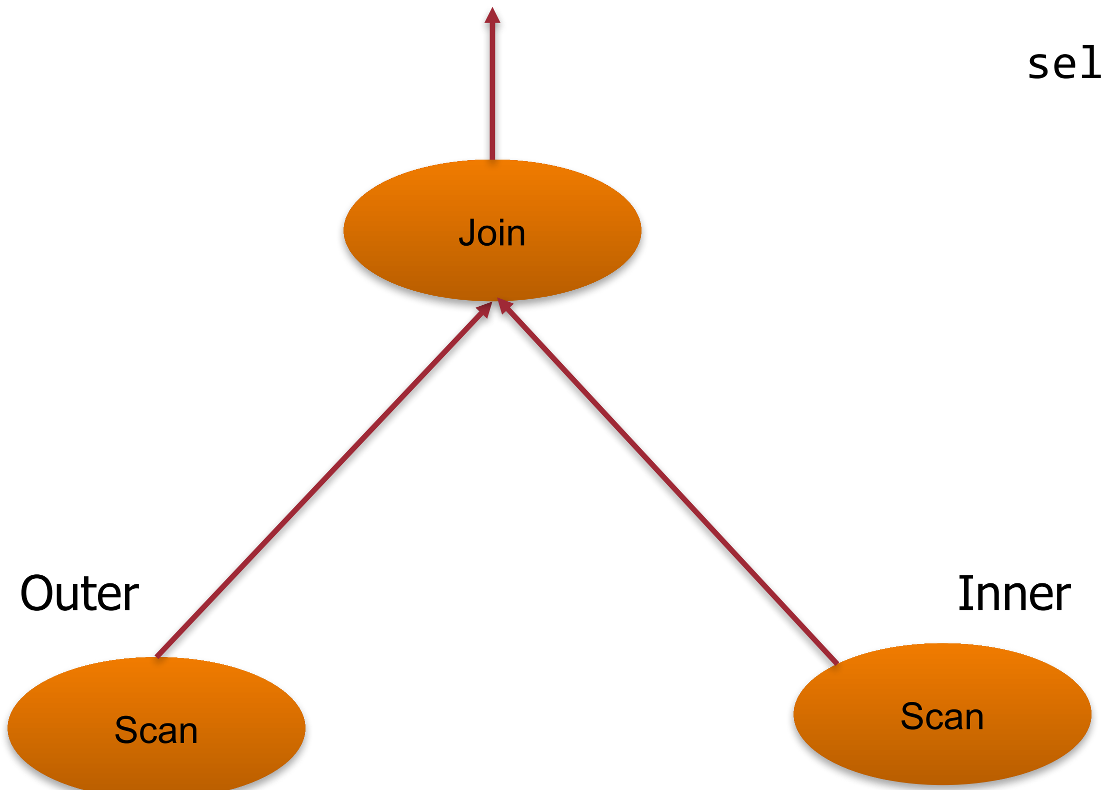
Example: “Exchange” operator



Optimized aggregation

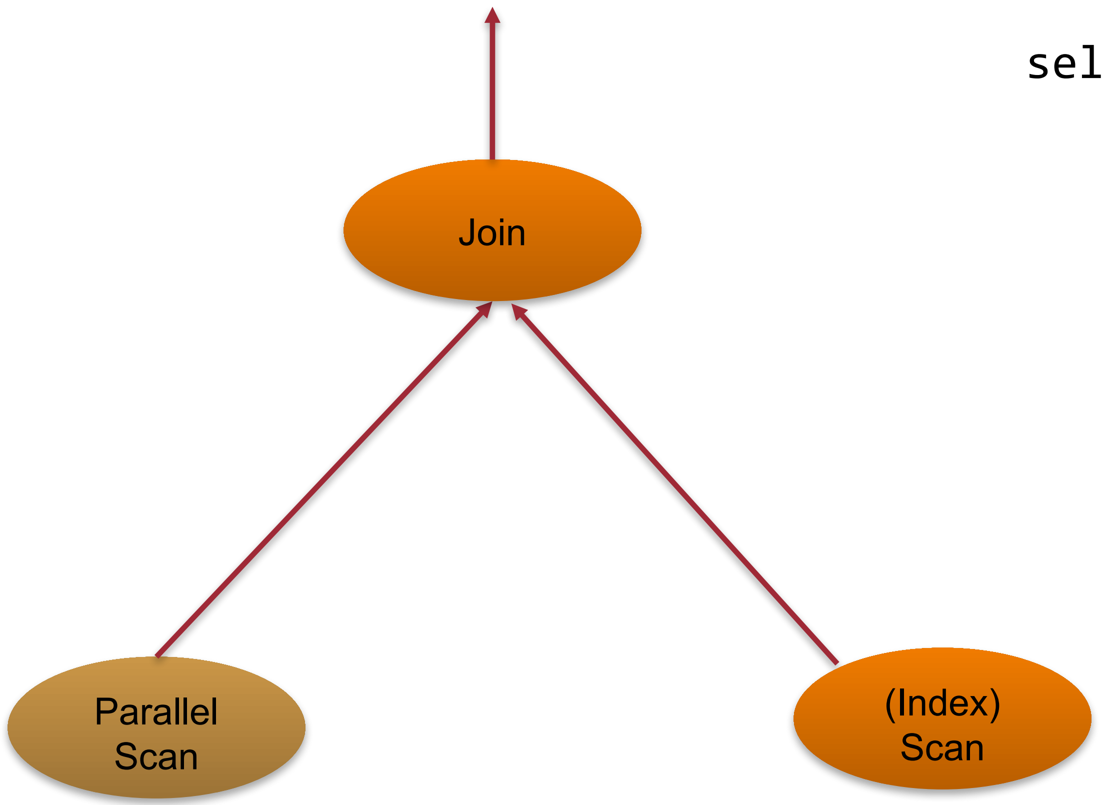


Parallel execution: Join



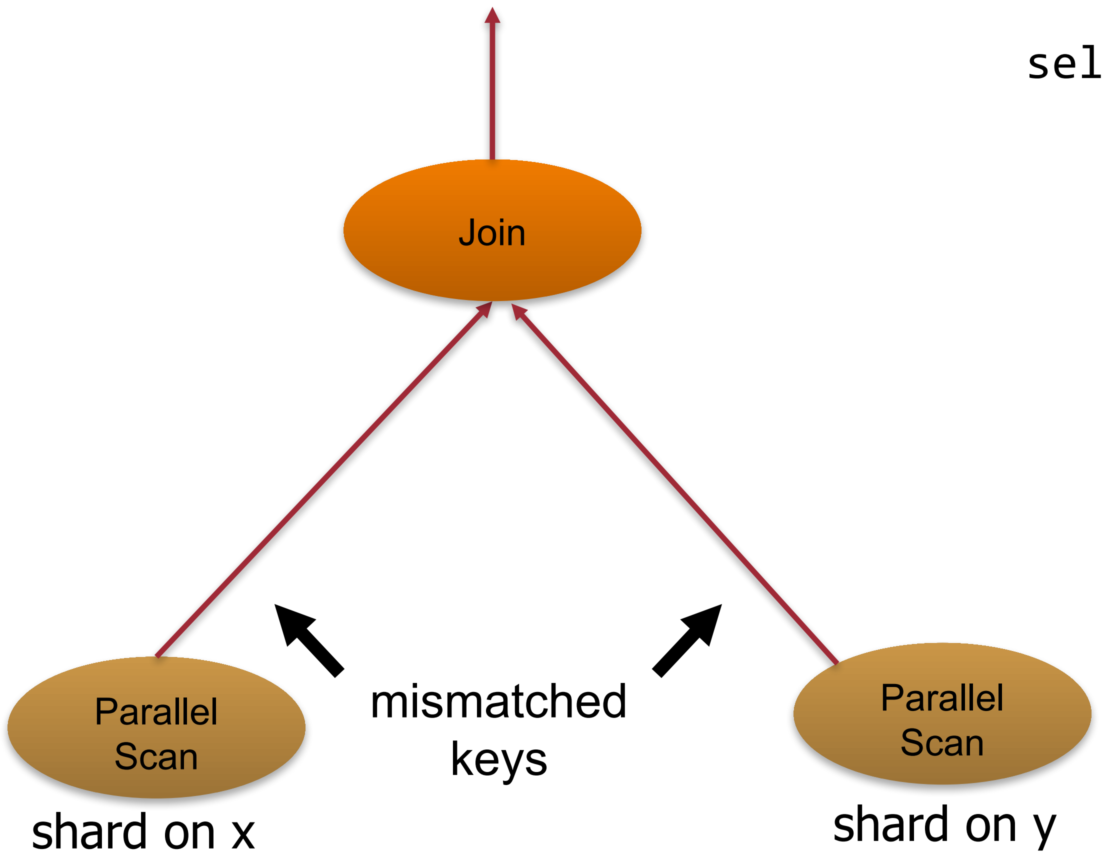

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: NLJoin



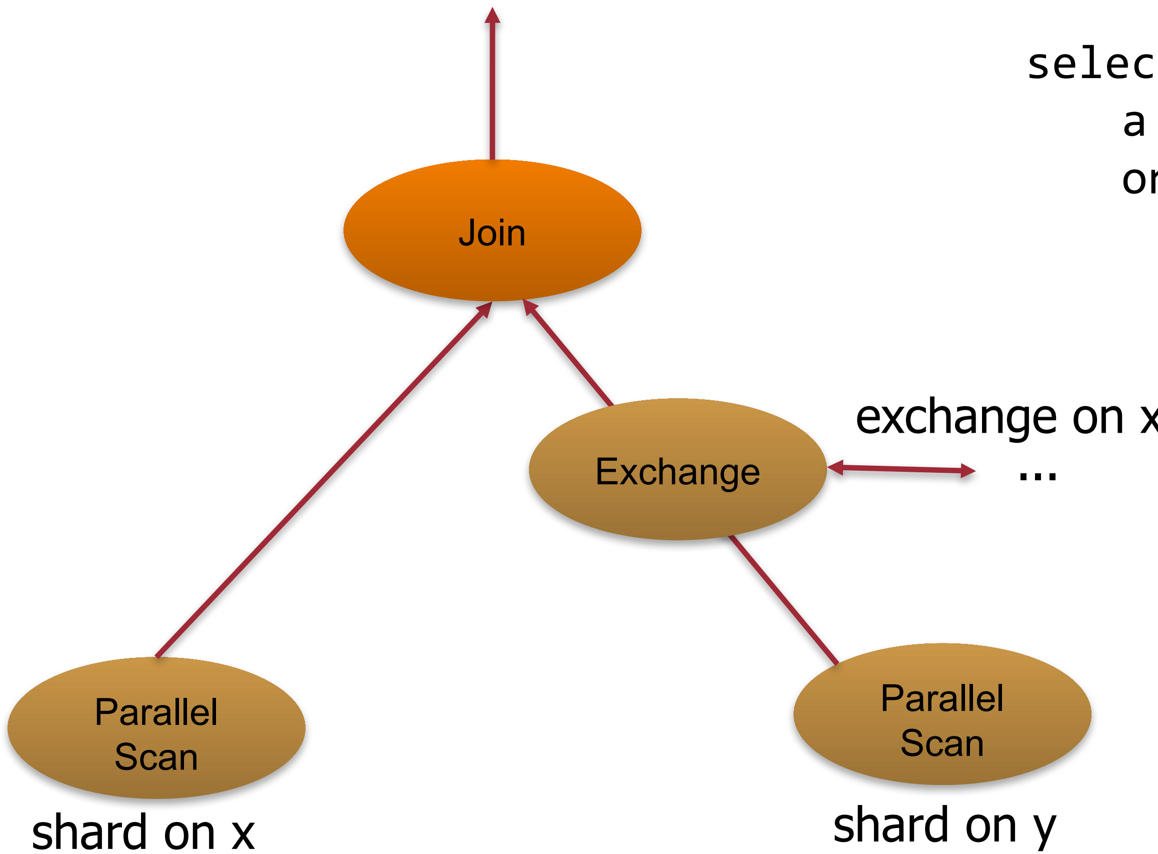

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: MergeJoin



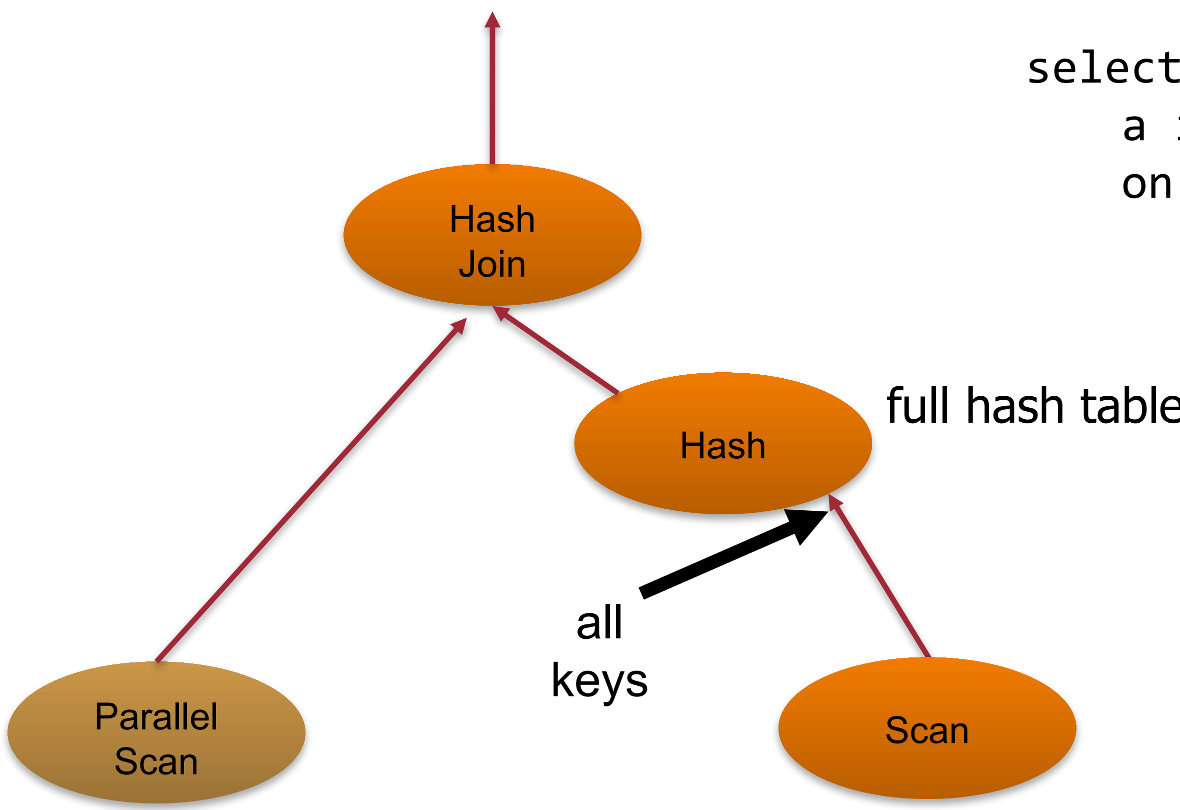

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: Join



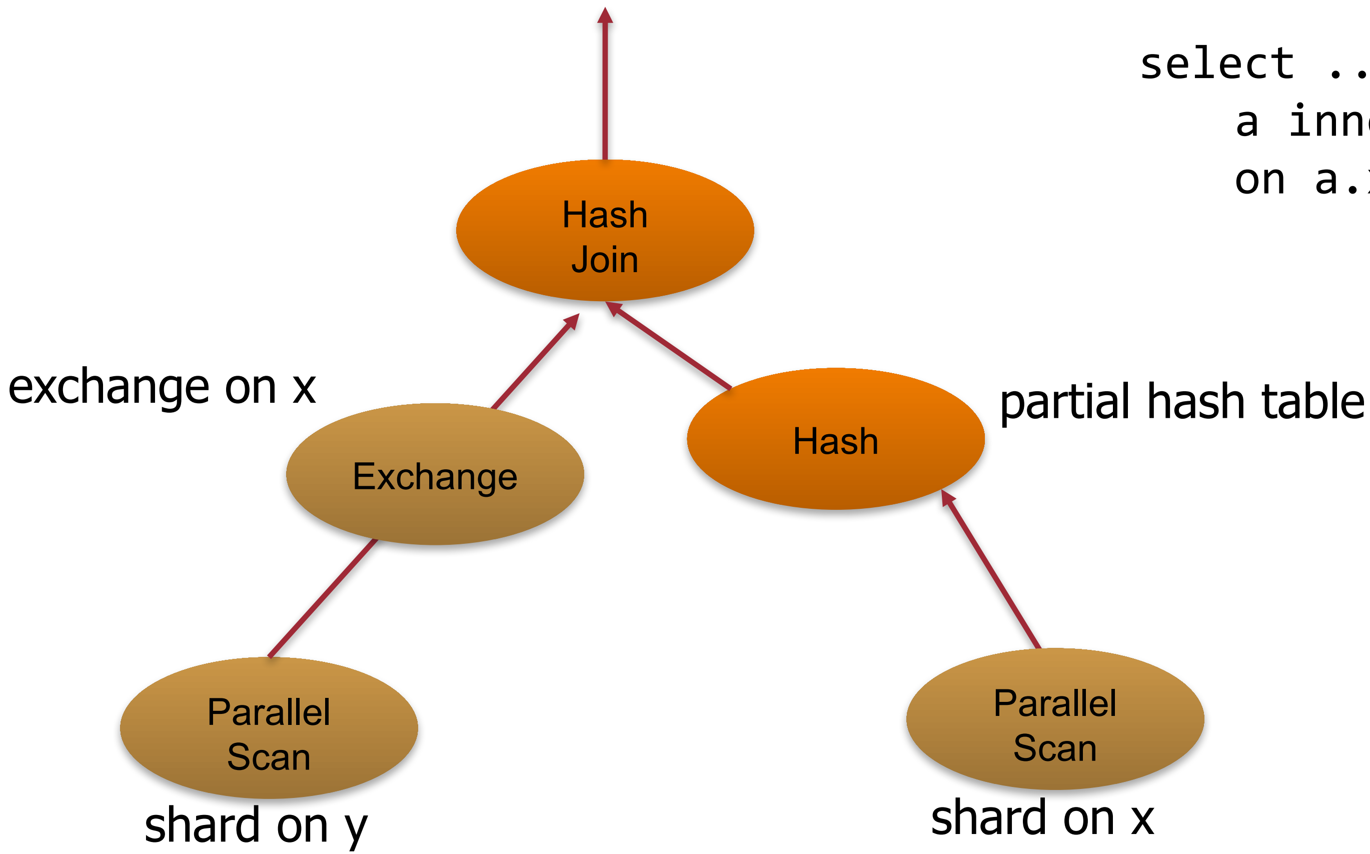

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: Hash Join (shared)



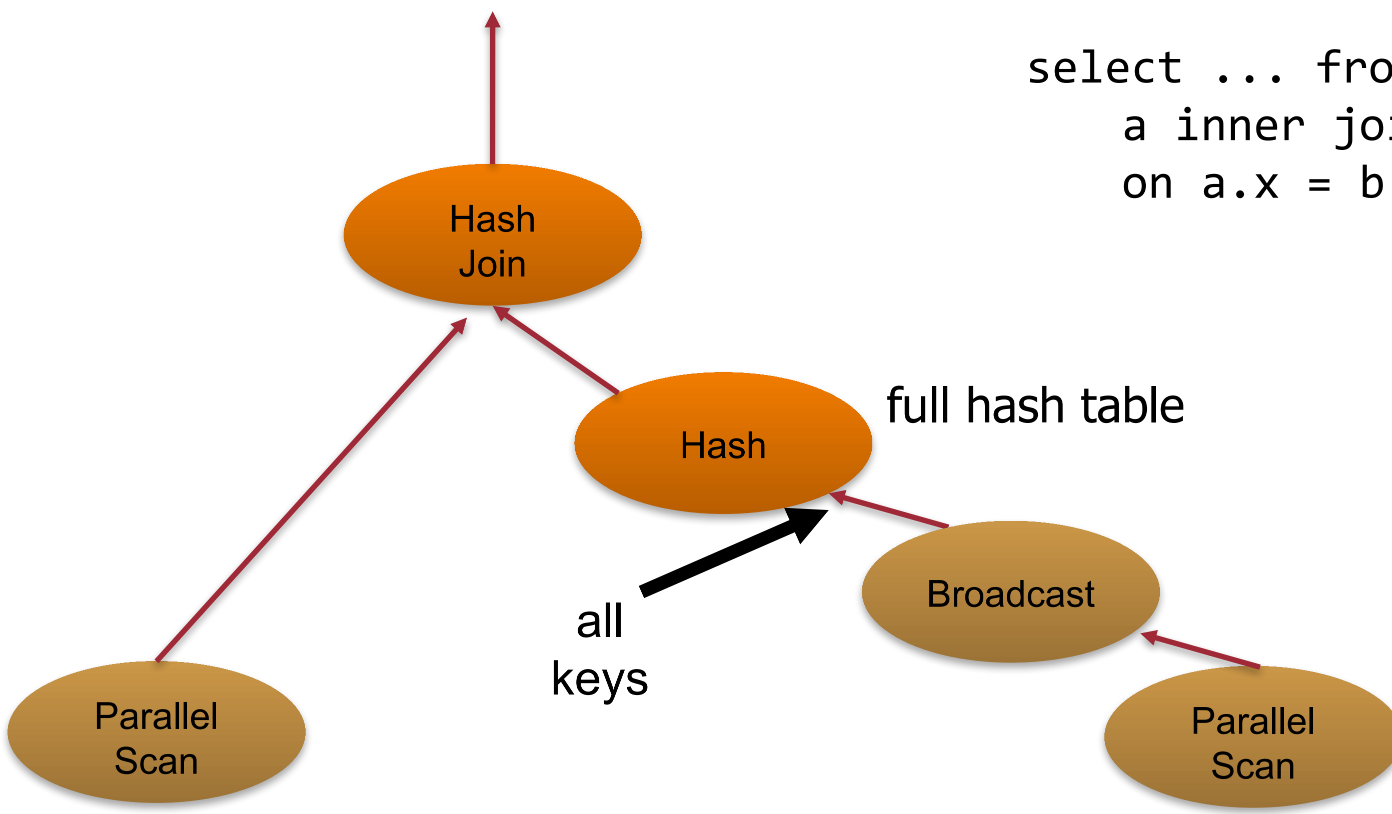

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: Hash Join (distributed)



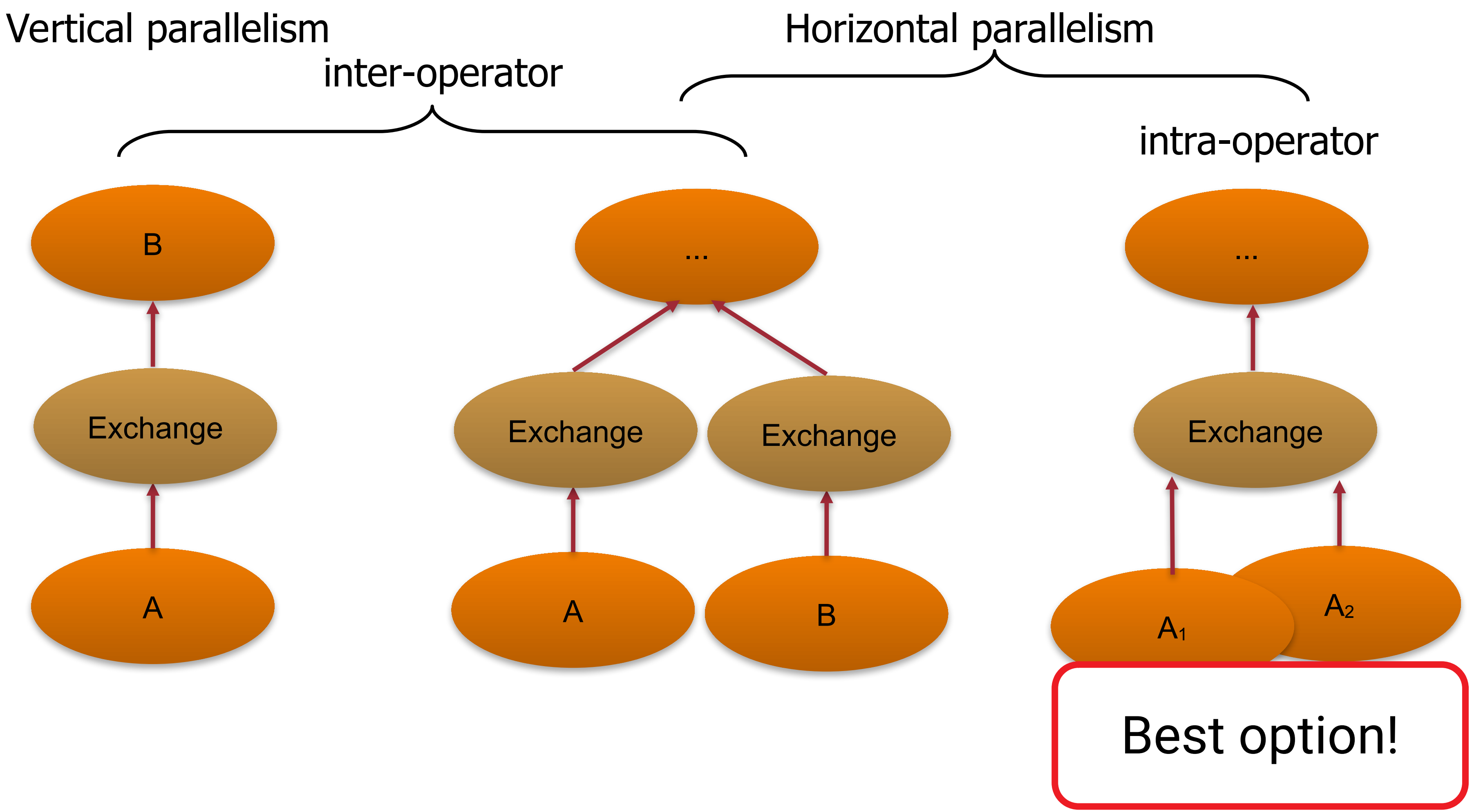

```
select ... from
    a inner join b
    on a.x = b.x
```

Parallel execution: Hash Join (distributed)




```
select ... from
   a inner join b
   on a.x = b.x
```

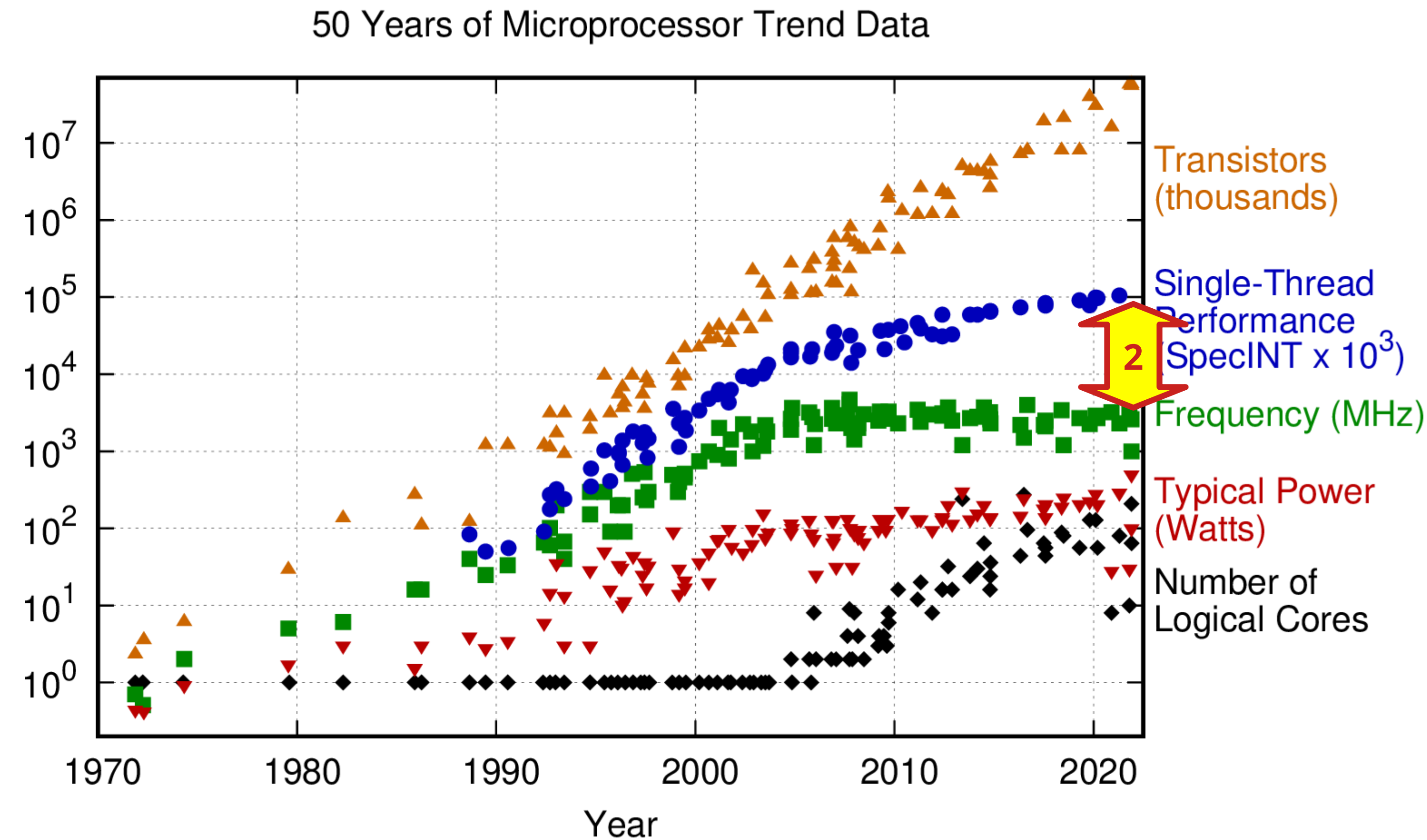
Consequences of Volcano "Exchange"



Consequences of Volcano "Exchange"

- Can easily retrofit existing database systems
- Parallelism is configured top-dow, but optimal parallelism is discovered bottom-up
  - Data distribution
- Adequate for a small number of cores

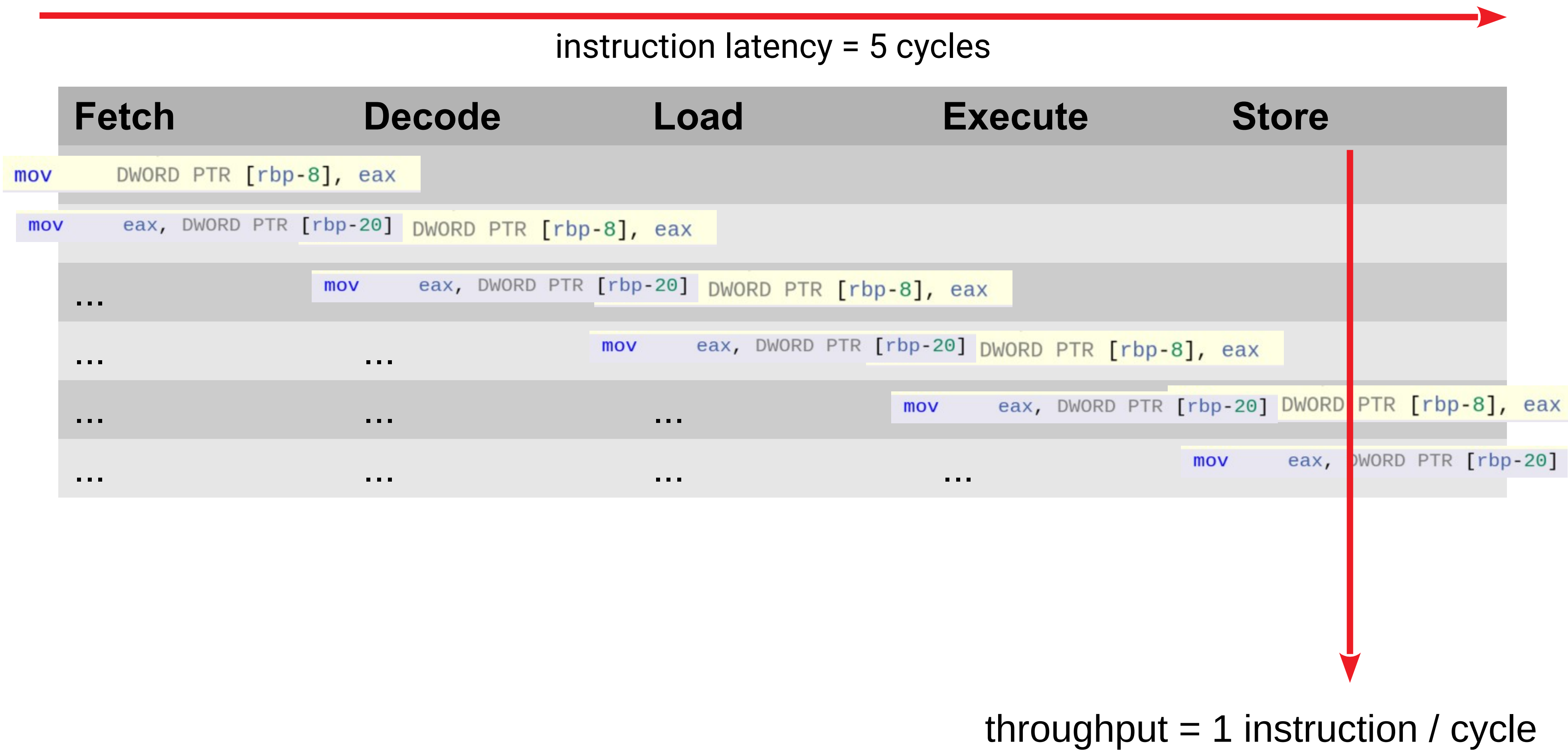
Instruction parallelism



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Source https://github.com/karlrupp/microprocessor-trend-data

Pipelining


instruction latency = 5 cycles
Fetch
Decode
Load
Execute
Store
mov DWORD PTR [rbp-8], eax
mov eax, DWORD PTR [rbp-20]
DWORD PTR [rbp-8], eax
...
throughput = 1 instruction / cycle

Pipelining

Fetch	Decode	Load	Execute	Store
`mov eax, DWORD PTR [rbp-20]`				
`add eax, eax`	`mov eax, DWORD PTR [rbp-20]`			
...	`add eax, eax`	`eax, DWORD PTR [rbp-20]`		
...	...	???	`mov eax, DWORD PTR [rbp-20]`	
...	...	...	stall	`mov eax, DWORD PTR [rbp-20]`
...	...	...	...	stall

- Data dependency:
  - Trying to load a value that has not yet been computed

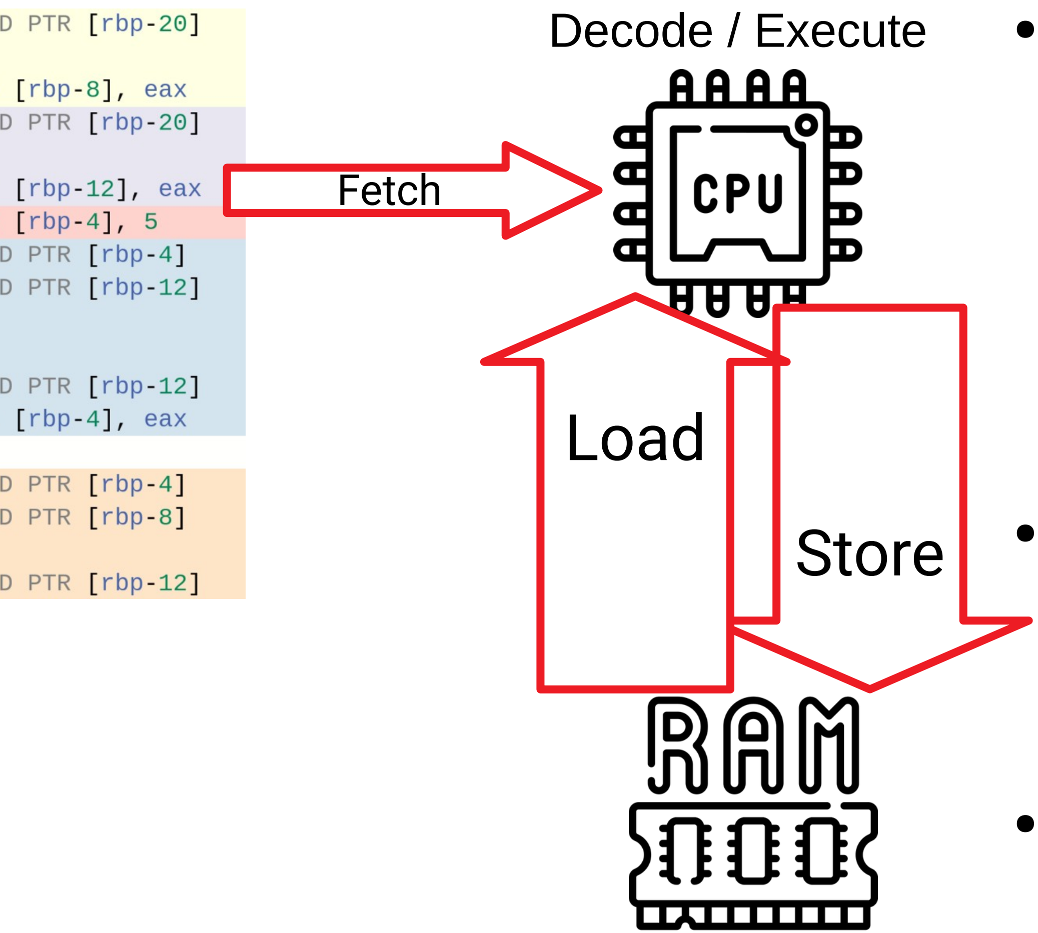
Pipelining

Fetch	Decode	Load	Execute	Store
`jns .L2`				
???	`jns .L2`			
stall	stall	`jns .L2`		
stall	stall	stall	`jns .L2`	
stall	stall	stall	stall	`jns .L2`
		stall	stall	stall

```
.L2:
        mov     edx, DWORD PTR [rbp-4]
```

- Control flow dependency:
  - Cannot predict the next instruction

SIMD



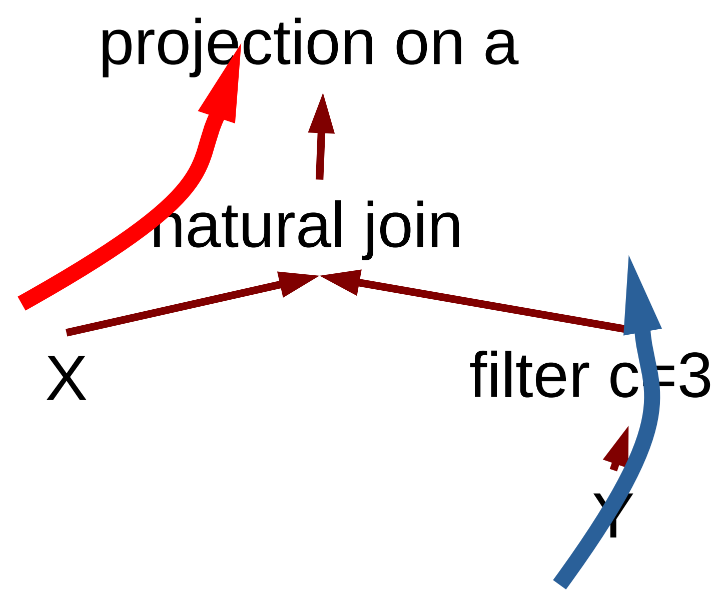
- Use wide registers that can fit vectors instead of scalars:
  - Example: Intel AVX512 → 512 bits
    - 64 byte vector
    - 32 shorts
    - 16 ints
    - …
- Load, execute, and store full vectors, or slices of vectors, in a single instruction
- Key technique in GPUs

Consequences of iteration

- Close to worst case scenario for data movement and parallelism! 😱
  - Poor locality → Impacts caching / NUMA
  - Short code segments interleaved with dereferencing though virtual pointers → Processor pipeline stalls
  - Computation on one value at a time → No SIMD
- Severely impacts <u>analytical</u> workloads!

Operator fusion

- Split the execution plan into pipelines
  - 1: scan Y → filter → hash
  - 2: scan X → join → projection
- Allows optimization step with code generation:
  - Optimized for pipelining and locality



Push model

- Data producer calls consumer:

```
for row in table:
    transform(row)
```

- Large overhead if there is a buffer between every two operators
- Viable with operator fusion
  - Buffer at the end of each pipeline

Chunked data

APACHE ARROW

- Iterate over “chunks”:
  - Records → Records of arrays
- Advantages of columnar layout: SIMD

b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

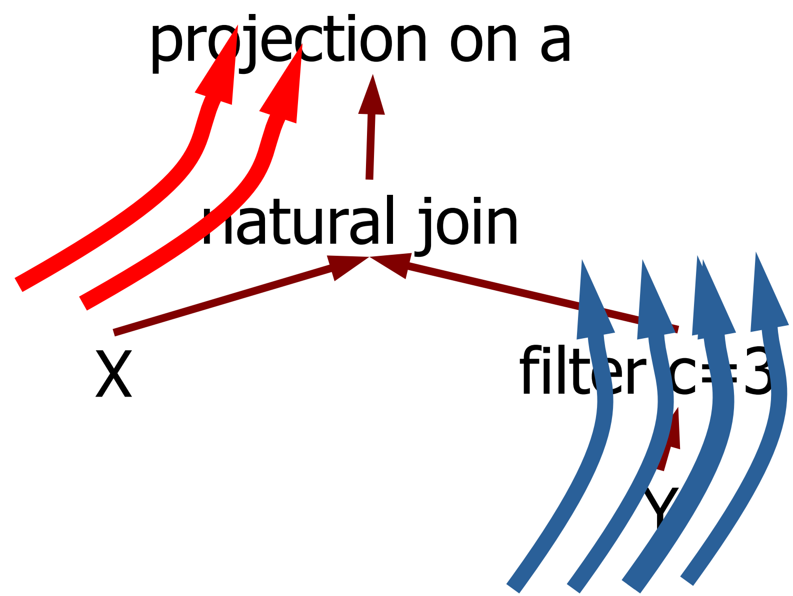
b	c
aaa	1
bbb	2
bbb	3
ccc	3
ddd	4

Chunked with operator fusion

- For each pipeline, generate a fused operator:
  - Outer loop waits for a chunk
    - Inner loop computes transformed chunk
  - Asynchronously delivers chunk
- Compile and optimize each pipeline:
  - Generate native code
  - Unroll inner loop, using SIMD instructions

Chunked with operator fusion

- Dynamically schedule multiple instances of each pipeline:
  - Depending on availability of input
  - Demand for output
  - Available resources



Consequences

- Ideal scenario for parallel processing:
  - Compiled fused operators → Simple and predictable control flow
  - Chunks → Good locality and SIMD
  - Pipeline scheduling → Dynamic multi-threading
  - Push model → Easier multi-thread scheduling

DuckDB

Conclusion

- There are a number of options for executing each query
- More options if we consider other data structures
- Varying performance:
  - Memory requirements
  - Number of iterations
  - Disk accesses
- What is the best one?
- How can it be discovered?