Data Processing on Modern Hardware

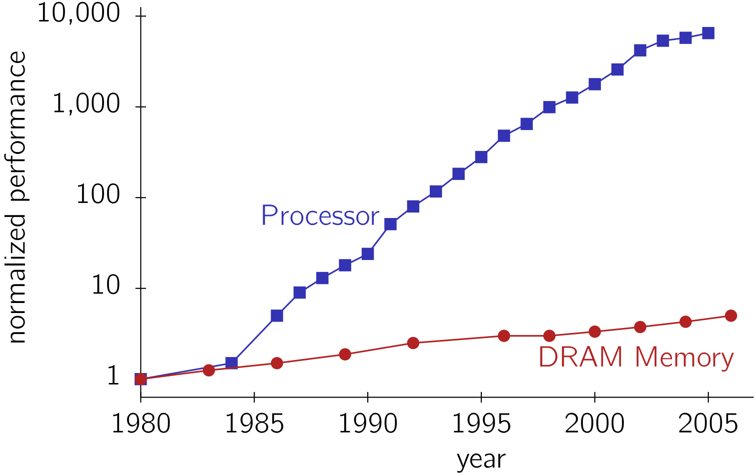
Jens Teubner, TU Dortmund, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2016

Part II

Cache Awareness

Hardware Trends



Source: Hennessy & Patterson, Computer Architecture, 4th Ed.

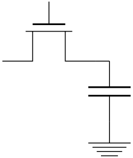
Hardware Trends

There is an increasing **gap** between CPU and memory speeds.

- Also called the **memory wall**.
- CPUs spend much of their time **waiting** for memory.

Memory ≠ Memory

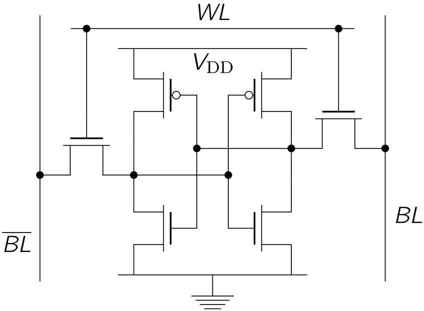
Dynamic RAM (DRAM)



- State kept in **capacitor**
- Leakage

  → **refreshing** needed

Static RAM (SRAM)



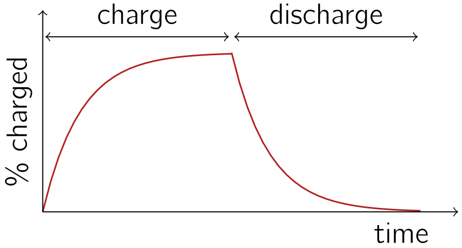
- **Bistable** latch (0 or 1)
- Cell state stable

  → no refreshing needed

DRAM Characteristics

Dynamic RAM is comparably **slow**.

- Memory needs to be **refreshed** periodically ($\approx$ every 64 ms).
- (Dis-)charging a capacitor takes time.



- DRAM cells must be addressed and capacitor outputs amplified.

Overall we're talking about $\approx$ 200 CPU cycles per access.

DRAM Characteristics

Under certain circumstances, DRAM **can** be reasonably fast.

- DRAM cells are physically organized as a 2-d array.
- The discharge/amplify process is done for an **entire row**.
- Once this is done, more than one word can be read out.

In addition,

- Several DRAM cells can be used in **parallel**.
  - $\rightarrow$ Read out even more words in parallel.

We can exploit that by using **sequential access patterns**.

SRAM Characteristics

SRAM, by contrast, can be very **fast**.

- Transistors actively drive output lines, access almost **instantaneous**.

But:

- SRAMs are significantly more expensive (chip space $\equiv$ money)

Therefore:

- Organize memory as a **hierarchy**.
- Small, fast memories used as **caches** for slower memory.

Memory Hierarchy

	technology	capacity	latency
CPU	SRAM	bytes	< 1 ns
L1 Cache	SRAM	kilobytes	≈ 1 ns
L2 Cache	SRAM	megabytes	< 10 ns
main memory	DRAM	gigabytes	70–100 ns
$\vdots$			
disk			

- Some systems also use a 3rd level cache.
- cf. Architecture & Implementation course

  $\rightarrow$ Caches resemble the buffer manager but are **controlled by hardware**

Principle of Locality

Caches take advantage of the **principle of locality**.

- 90 % execution time spent in 10 % of the code.
- The **hot set** of data often fits into caches.

Spatial Locality:

- Code often contains loops.
- Related data is often spatially close.

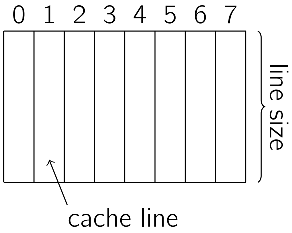
Temporal Locality:

- Code may call a function repeatedly, even if it is not spatially close.
- Programs tend to re-use data frequently.

CPU Cache Internals

To guarantee speed, the **overhead** of caching must be kept reasonable.

- Organize cache in **cache lines**.
- Only load/evict **full cache lines**.
- Typical **cache line size**: 64 bytes.



- The organization in cache lines is consistent with the principle of (spatial) locality.
- Block-wise transfers are well-supported by DRAM chips.

Memory Access

On every memory access, the CPU checks if the respective **cache line** is already cached.

Cache Hit:

- Read data directly from the cache.
- No need to access lower-level memory.

Cache Miss:

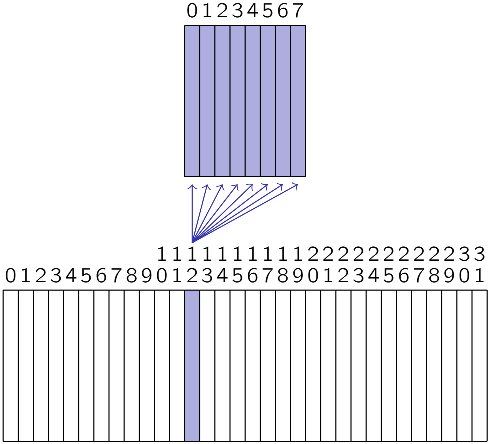
- Read full cache line from lower-level memory.
- Evict some cached block and replace it by the newly read cache line.
- CPU **stalls** until data becomes available.[1]

[1] Modern CPUs support out-of-order execution and several in-flight cache misses.

Block Placement: Fully Associative Cache

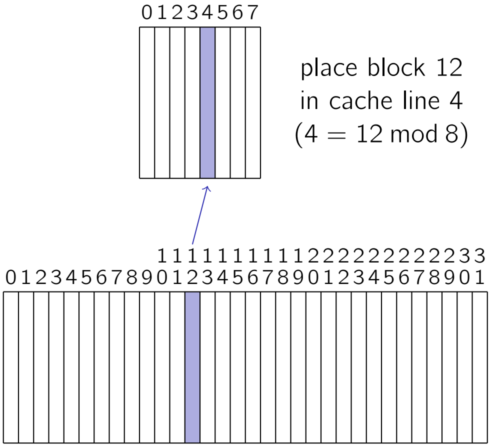
In a **fully associative** cache, a block can be loaded into **any** cache line.

- Offers freedom to block replacement strategy.
- Does not scale to large caches
  - → 4 MB cache, line size: 64 B: 65,536 cache lines.
- Used, *e.g.*, for small TLB caches.



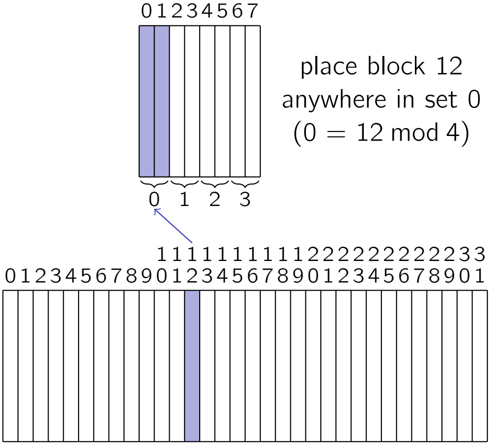
In a **direct-mapped** cache, a block has only one place it can appear in the cache.

- **Much** simpler to implement.
- Easier to make **fast**.
- Increases the chance of **conflicts**.

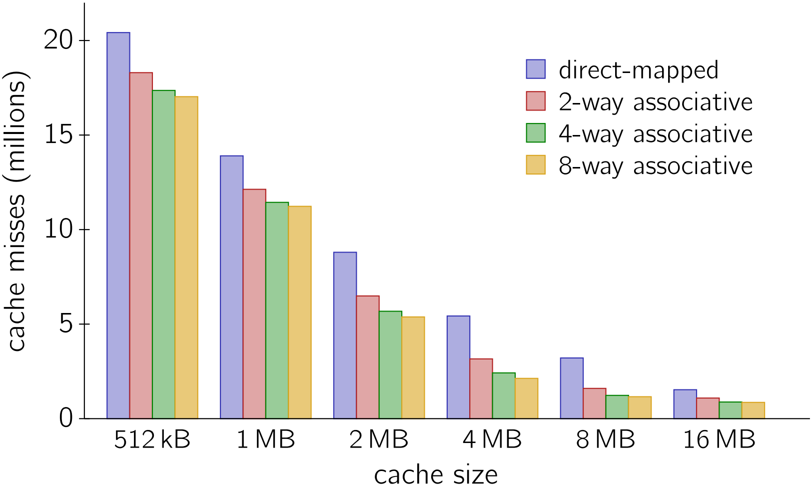


A compromise are **set-associative** caches.

- Group cache lines into **sets**.
- Each memory block maps to one set.
- Block can be placed anywhere **within** a set.
- Most processor caches today are set-associative.



Effect of Cache Parameters

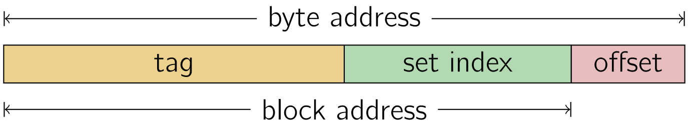


Source: Ulrich Drepper. What Every Programmer Should Know About Memory

Block Identification

A **tag** associated with each cache line identifies the memory block currently held in this cache line.

The **tag** can be derived from the **memory address**.



Example: Intel Q6700 (Core 2 Quad)

- Total cache size: **4 MB** (per 2 cores).
- Cache line size: **64 bytes**.
  - → 6-bit offset ($2^6 = 64$)
  - → There are 65,536 cache lines in total (4 MB ÷ 64 bytes).
- Associativity: **16-way set-associative**.
  - → There are 4,096 sets ($65,536 \div 16 = 4,096$).
  - → 12-bit set index ($2^{12} = 4,096$).
- Maximum physical address space: **64 GB**.
  - → 36 address bits are enough (2^{36} bytes = 64 GB)
  - → 18-bit tags ($36 - 12 - 6 = 18$).

tag	set index	offset
18 bit	12 bit	6 bit

Block Replacement

When bringing in new cache lines, an existing entry has to be **evicted**.

Different strategies are conceivable (and meaningful):

Least Recently Used (LRU)

- Evict cache line whose last access is longest ago.
  $\rightarrow$ Least likely to be needed any time soon.

First In First Out (FIFO)

- Behaves often similar like LRU.
- But easier to implement.

Random

- Pick a random cache line to evict.
- Very simple to implement in hardware.

Replacement has to be decided **in hardware** and **fast**.

What Happens on a Write?

To implement memory **writes**, CPU makers have two options:

Write Through

- Data is directly written to lower-level memory (and to the cache).
  - → Writes will **stall the CPU**.[2]
  - → Greatly simplifies **data coherency**.

Write Back

- Data is only written into the cache.
- A **dirty** flag marks modified cache lines (Remember the status field.)
  - → May reduce traffic to lower-level memory.
  - → Need to write on eviction of dirty cache lines.

Modern processors usually implement **write back**.

---

[2] **Write buffers** can be used to overcome this problem.

Putting it all Together

To compensate for **slow memory**, systems use **caches**.

- DRAM provides **high capacity**, but **long latency**.
- SRAM has **better latency**, but **low capacity**.
- Typically multiple levels of caching (memory hierarchy).
- Caches are organized into **cache lines**.
- **Set associativity**: A memory block can only go into a small number of cache lines (most caches are set-associative).

Systems will benefit from **locality**.

- Affects data **and** code.

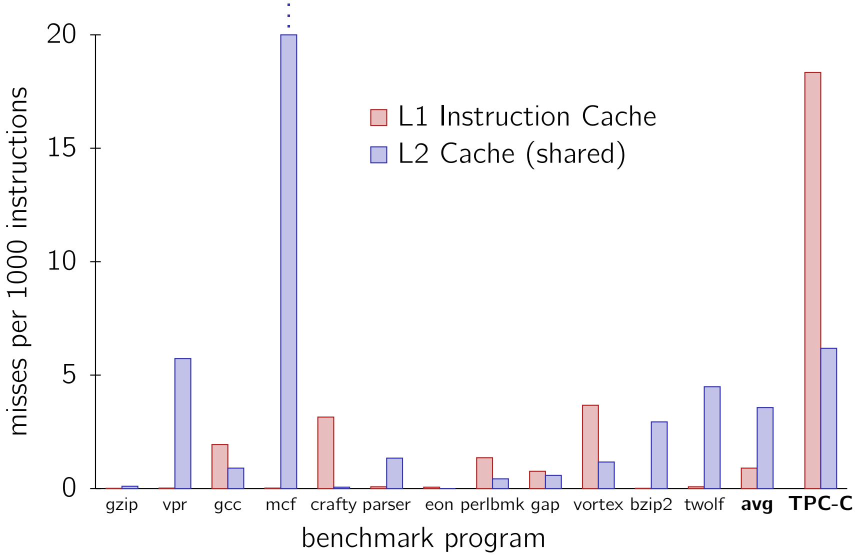
Example: AMD Opteron

Example: AMD Opteron, 2.8 GHz, PC3200 DDR SDRAM

- L1 cache: separate data and instruction caches, each 64 kB, 64 B cache lines, 2-way set-associative
- L2 cache: shared cache, 1 MB, 64 B cache lines, 16-way set-associative, pseudo-LRU policy
- L1 hit latency: 2 cycles
- L2 hit latency: 7 cycles (for first word)
- L2 miss latency: 160–180 cycles (20 CPU cycles + 140 cy DRAM latency (50 ns) + 20 cy on mem. bus)
- L2 cache: write-back
- 40-bit virtual addresses

Source: Hennessy & Patterson. Computer Architecture—A Quantitative Approach.

Performance (SPECint 2000)



Assessment

✎ **Why do database systems show such poor cache behavior?**

Data Caches

How can we improve data cache usage?

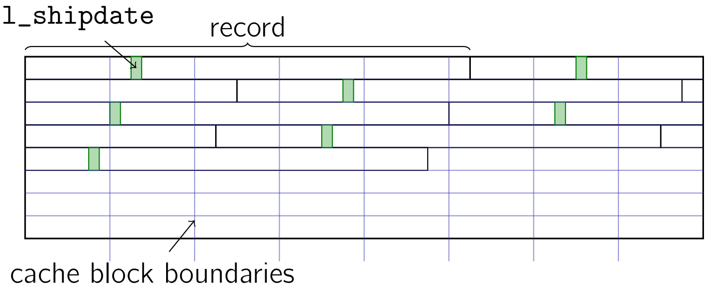
Consider, *e.g.*, a selection query:

```
SELECT COUNT(*)
  FROM lineitem
 WHERE l_shipdate = "2009-09-26"
```

- This query typically involves a **full table scan**.

Table Scans (NSM)

Tuples are represented as **records** stored sequentially on a database page.

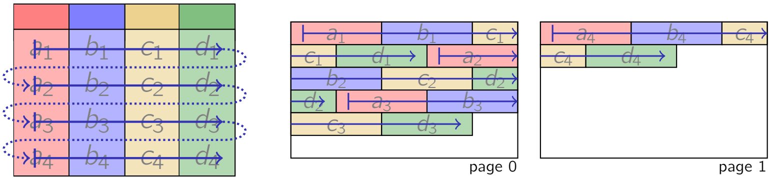


- With every access to a `l_shipdate` field, we load a large amount of **irrelevant** information into the cache.
- Accesses to slot directories and variable-sized tuples incur additional trouble.

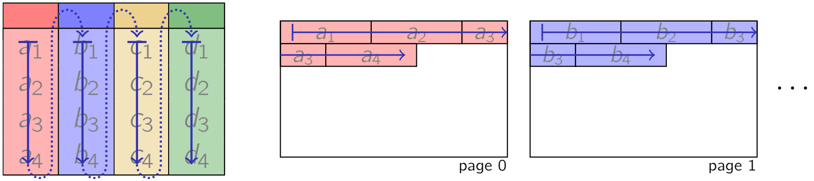
Row-Wise vs. Column-Wise Storage

Remember the "Architecture & Implementation" course?

The *n*-ary storage model (NSM, row-wise storage) is not the only choice.



Column-wise storage (decomposition storage model, DSM):



Column-Wise Storage

- All data loaded into caches by a "`l_shipdate` scan" is now actually relevant for the query.
  - $\rightarrow$ Less data has to be fetched from memory.
  - $\rightarrow$ Amortize cost for fetch over more tuples.
  - $\rightarrow$ If we're really lucky, the full (`l_shipdate`) data might now even fit into caches.
- The same arguments hold, by the way, also for disk-based systems.
- Additional benefit: Data compression might work better.

↗ Copeland and Khoshafian. A Decomposition Storage Model. *SIGMOD 1985*.

MonetDB: Binary Association Tables

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

oid	NAME	AGE	SEX
o_1	John	34	m
o_2	Angelina	31	f
o_3	Scott	35	m
o_4	Nancy	33	f

$\rightarrow$

oid	NAME
o_1	John
o_2	Angelina
o_3	Scott
o_4	Nancy

oid	AGE
o_1	34
o_2	31
o_3	35
o_4	33

oid	SEX
o_1	m
o_2	f
o_3	m
o_4	f

- Each column yields one **binary association table (BAT)**.
- **Object identifiers** (`oid`s) identify matching entries (BUNs).
- Oftentimes, `oid`s can be implemented as **virtual** `oid`s (`void`s).
  $\rightarrow$ Not explicitly materialized in memory.

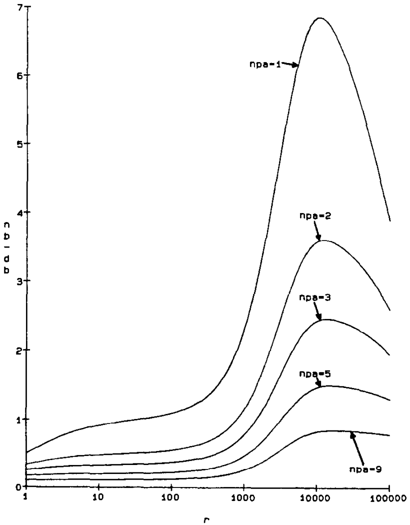
NSM vs. DSM Trade-Offs

Tuple recombination can cause considerable cost.

- Need to perform **many joins**.
- Workload-dependent trade-off.

→ MonetDB: **positional joins** (thanks to `void` columns)

Figure 2 Varying The Number Of Projected Attributes



Copeland and Khoshafian. A Decomposition Storage Model. *SIGMOD 1985.*

Column Stores in Commercial DBMSs

Commercial databases have just recently announced column-store extensions to their engines:

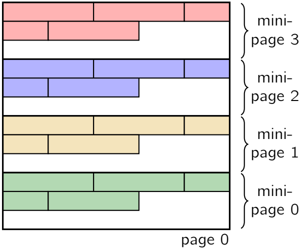
- **Microsoft SQL Server**:
  - Represented as “Column Store Indexes”
  - Available since SQL Server 11
  - see Larson *et al.*, SIGMOD 2011
- **IBM DB2**:
  - IBM announced DB2 “BLU Accelerator” last week, a column store that is going to ship with DB2 10.5.
  - BLU stands for “Blink Ultra”; Blink was developed at IBM Almaden (↗ Raman et al., *ICDE 2008*).

PAX: Another Alternative

A hybrid approach is the **PAX (Partition Attributes Accross)** layout:

- Divide each page into **minipages**.
- Group attributes into them.

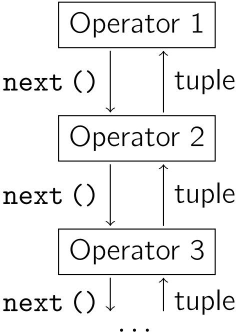
↗ Ailamaki *et al.* Weaving Relations for Cache Performance. *VLDB 2001*.



Processing Characteristics

Most systems implement the **Volcano iterator model**:

- Operators request tuples from their input using `next ()`.
- Data is processed **tuple at a time**.
- **“pipelining”**
- Each operator keeps its own **state**.
- ↗ DB implementation course



Tuple-At-A-Time Processing

Consequences:

- All operators in a plan run **tightly interleaved**.
  - → Their **combined** instruction footprint may be large.
  - → **Instruction cache misses**.
- Operators constantly call each other's functionality.
  - → Large **function call overhead**.
- The combined **state** may be too large to fit into caches.
  - *E.g.*, hash tables, cursors, partial aggregates.
  - → **Data cache misses**.

Example: TPC-H On MySQL

Example: Query Q1 from the TPC-H benchmark on MySQL.

```
SELECT l_returnflag, l_linestatus, SUM (l_quantity) AS sum_qty,
       SUM(l_extendedprice) AS sum_base_price,
       SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
       SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge,
       AVG(l_quantity) AS avg_qty, AVG(l_extendedprice) AS avg_price,
       AVG(l_discount) AS avg_disc, COUNT(*) AS count_order
  FROM lineitem
 WHERE l_shipdate <= DATE '1998-09-02'
 GROUP BY l_returnflag, l_linestatus
```

- **Scan query** with **arithmetics** and a bit of aggregation.

Results taken from Peter Boncz, Marcin Zukowski, Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR 2005*.

time [sec]	calls	instr./call	IPC	function name
11.9	846M	6	0.64	ut_fold_ulint_pair
8.5	0.15M	27K	0.71	ut_fold_binary
5.8	77M	37	0.85	memcpy
3.1	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
3.0	6M	247	0.83	row_search_for_mysql
2.9	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
2.6	108M	11	0.60	rec_get_bit_field_1
2.5	6M	213	0.61	row_sel_store_mysql_rec
2.4	48M	25	0.52	rec_get_nth_field
2.4	60	19M	0.69	ha_print_info
2.4	5.9M	195	1.08	end_update
2.1	11M	89	0.98	field_conv
2.0	5.9M	16	0.77	Field_float::val_real
1.8	5.9M	14	1.07	Item_field::val
1.5	42M	17	0.51	row_sel_field_store_in_mysql
1.4	36M	18	0.76	buf_frame_align
1.3	**17M**	**38**	**0.80**	**Item_func_mul::val**
1.4	25M	25	0.62	pthread_mutex_unlock
1.2	206M	2	0.75	hash_get_nth_cell
1.2	25M	21	0.65	mutex_test_and_set
1.0	102M	4	0.62	rec_get_1byte_offs_flag
1.0	53M	9	0.58	rec_1_get_field_start_offs
0.9	42M	11	0.65	rec_get_nth_field_extern_bit
1.0	**11M**	**38**	**0.80**	**Item_func_minus::val**
0.5	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Observations

Observations:

- Only **single tuple** processed in each call; **millions of calls**.
- Only **10 % of the time** spent on actual query task.
- Very low **instructions-per-cycle** (IPC) ratio.

Further:

- Much time spent on **field access** (*e.g.*, rec_get_nth_field ()).
  - NSM $\leadsto$ polymorphic operators.
- Single-tuple functions hard to optimize (by compiler).
  - $\rightarrow$ Low instructions-per-cycle ratio.
  - $\rightarrow$ Vector instructions (SIMD) hardly applicable.
- Function call overhead.
  - $\frac{38\,\text{instr.}}{0.8\,\frac{\text{instr.}}{\text{cycle}}} = 48\,\text{cycles}$ **vs.** 3 instr. for load/add/store assembly.

Operator-At-A-Time Processing

MonetDB: **operator-at-a-time processing**.

- Operators consume and produce **full columns**.
- Each (sub-)result is **fully materialized** (in memory).
- **No** pipelining (rather a sequence of statements).
- Each operator runs exactly once.

Example:

```
sel_age   := people_age.select(30, nil);
sel_id    := sel_age.mirror().join(people_age);
sel_name  := sel_age.mirror().join(people_name);
tmp       := [-](sel_age, 30);
sel_bonus := [*](50, tmp);
```

Operator-At-A-Time Processing

Few function calls; **extremely tight loops** when iterating over tuples.

Example: `batval_int_add(···)` (impl. of `[+](int,BAT[any,int])`)

```
⋮
if (vv != int_nil) {
    for (; bp < bq; bp++, bnp++) {
        REGISTER int bv = *bp;
        if (bv != int_nil) {
            bv = (int) OP(bv,+,vv);
        }
        *bnp = bv;
    }
} else {
    for (; bp < bq; bp++, bnp++) {
        *bnp = vv;
    }
}
⋮
```

Tight Loops

These tight loops

- conveniently **fit into instruction caches**,
- can be **optimized** effectively by modern compilers,
  - $\rightarrow$ **loop unrolling**
  - $\rightarrow$ **vectorization** (use of SIMD instructions)
- can leverage modern CPU features (**hardware prefetching**).

Function calls are now **out of the critical code path**.

Note also:

- **No** per-tuple field extraction or type resolution.
  - **Operator specialization**, *e.g.*, for every possible type.
  - Implemented using **macro expansion**.
  - Possible due to column-based storage.

result size	time [ms]	bandwidth [MB/s]	MIL statement
5.9M	127	352	`s0 := select(l_shipdate, ···).mark();`
5.9M	134	505	`s1 := join(s0, l_returnag);`
5.9M	134	506	`s2 := join(s0, l_linestatus);`
5.9M	235	483	`s3 := join(s0, l_extprice);`
5.9M	233	488	`s4 := join(s0, l_discount);`
5.9M	232	489	`s5 := join(s0, l_tax);`
5.9M	134	507	`s6 := join(s0, l_quantity);`
5.9M	290	155	`s7 := group(s1);`
5.9M	329	136	`s8 := group(s7, s2);`
4	0	0	`s9 := unique(s8.mirror());`
5.9M	206	440	`r0 := [+](1.0, s5);`
5.9M	210	432	`r1 := [-](1.0, s4);`
5.9M	274	498	`r2 := [*](s3, r1);`
5.9M	274	499	`r3 := [*](s12, r0);`
4	165	271	`r4 := {sum}(r3, s8, s9);`
4	165	271	`r5 := {sum}(r2, s8, s9);`
4	163	275	`r6 := {sum}(s3, s8, s9);`
4	163	275	`r7 := {sum}(s4, s8, s9);`
4	144	151	`r8 := {sum}(s6, s8, s9);`
4	112	196	`r9 := {count}(s7, s8, s9);`
	3,724	365	

Source: Boncz *et al.*, MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR 2005.*

Tuple-At-A-Time vs. Operator-At-A-Time

The **operator-at-a-time model** is a two-edged sword:

- ☺ Cache-efficient with respect to **code** and **operator state**.
- ☺ Tight loops, optimizable code.

- ☹ **Data** won't fully fit into cache.
  - → Repeated scans will fetch data **from memory** over and over.
  - → Strategy falls apart when intermediate results no longer fit into main memory.

Can we aim for the **middle ground** between the two extremes?

tuple-at-a-time ⟵⟶ **operator-at-a-time**

↑

X100 vectorized execution

Vectorized Execution Model

Idea:

- Use Volcano-style iteration,

but:

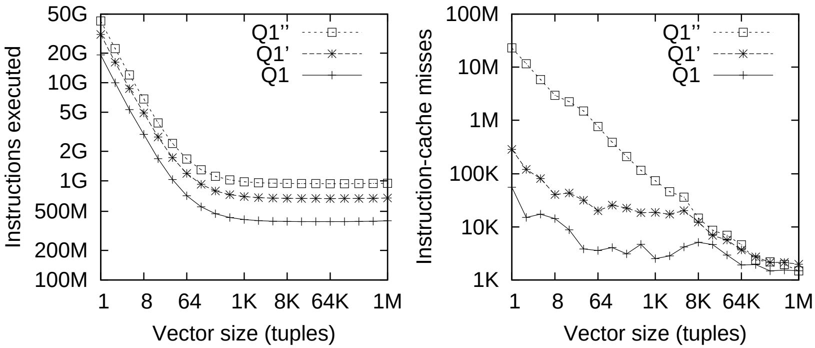
- for each `next ()` call **return a large number of tuples**
  → a "vector" in MonetDB/X100 terminology.

Choose vector size

- **large enough** to compensate for iteration overhead (function calls, instruction cache misses, . . .), but
- **small enough** to not thrash data caches.

✎ **Will there be such a vector size?** (Or will caches be thrashed long before iteration overhead is compensated?)

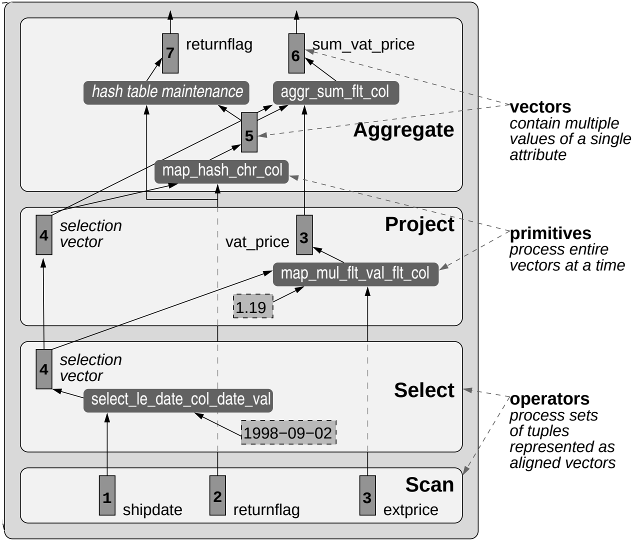
Vector Size ↔ Instruction Cache Effectiveness



Source: M. Zukowski. Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD thesis, CWI Amsterdam. 2009.

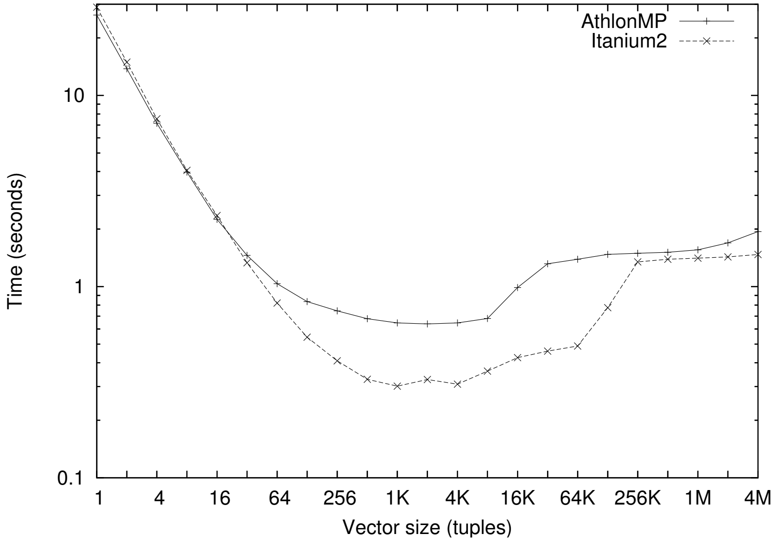
- Vectorized execution quickly compensates for iteration overhead.
- 1000 tuples should conveniently fit into caches.

Vectorized Execution in MonetDB/X100



Source: M. Zukowski. Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD thesis, CWI Amsterdam. 2009.

Effect on Query Execution Time



Boncz *et al.* MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR 2005.*

Comparison of Execution Models

Overview over discussed execution models:

execution model	tuple	operator	vector
query plans	simple	complex	simple
instr. cache utilization	poor	extremely good	very good
function calls	many	extremely few	very few
attribute access	complex	direct	direct
most time spent on	interpretation	processing	processing
CPU utilization	poor	good	very good
compiler optimizations	limited	applicable	applicable
materialization overhead	very cheap	expensive	cheap
scalability	good	limited	good

source: M. Zukowski. Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD thesis, CWI Amsterdam. 2009.

Vectorized Execution in SQL Server 11

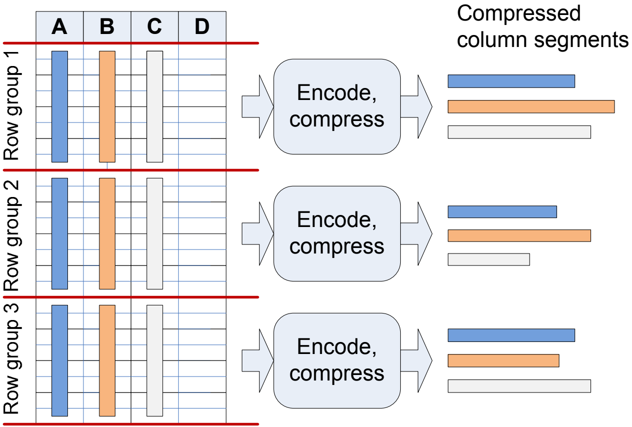
Microsoft SQL Server supports vectorized ("batched" in MS jargon) execution since version 11.

- Storage via new **column-wise index**.
  - → Includes **compression** and **prefetching improvements**.
- New operators with **batch-at-a-time processing**.
  - → Can combine row- and batch-at-a-time operators in one plan.
  - → CPU-optimized implementations.

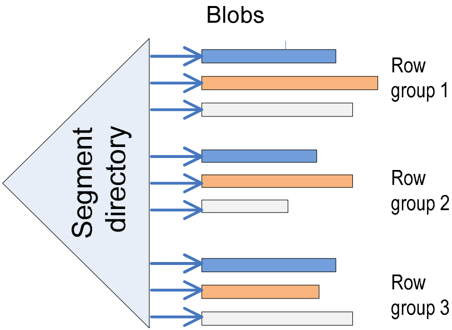
↗ Per-Åke Larson *et al.* SQL Server Column Store Indexes. *SIGMOD 2011*.

Column-Wise Index Storage

- Tables divided into **row groups** ($\approx$ 1 million rows)
- Each group, each column **compressed** independently.



Segment Organization



- **Segment directory** keeps track of segments.
- Segments are stored as **BLOBs** ("binary large objects")
  - ⤳ Re-use existing SQL Server functionality.
- Statistics (min/max values) for each segment.

I/O Optimizations

Column-store indexes are designed for **scans**.

- **Compression** (RLE, bit packing, dictionary encoding)
  - → Re-order row groups for best compression.
- Segments are forced to be **contiguous on disk**.
  - → Unlike typical page-by-page storage.
  - → Pages and segments are automatically **prefetched**.

data set	uncompressed	column-store idx	ratio
cosmetics	1,302	88.5	14.7
SQM	1,431	166	8.6
Xbox	1,045	202	5.2
MSSales	642,000	126,000	5.1
Web Analytics	2,560	553	4.6
Telecom	2,905	727	4.0

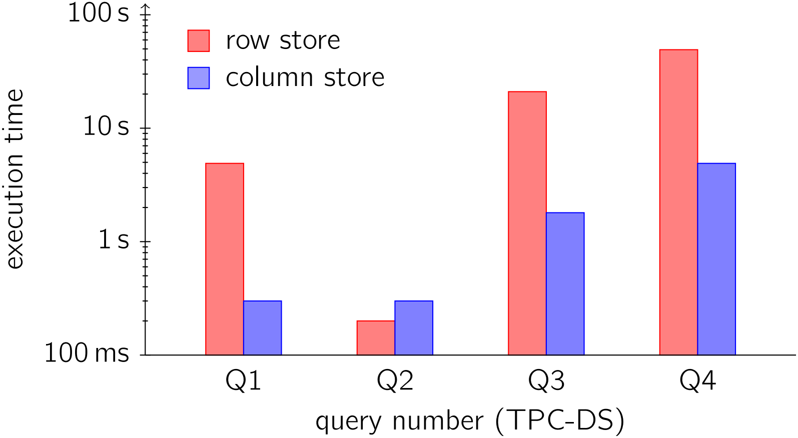
Batched Execution

Similar to the X100/Vectorwise execution model, **batch operators** in SQL Server can process batches of tuples at once.

- Can mix batch- and row-based processing in one plan.
- Typical pattern:
  - → Scan, pre-filter, project, aggregate data early in the plan using **batch operators**.
  - → **Row operators** may be needed to finish the operation.
- Good for scan-intensive workloads (OLAP) , **not** for point queries (OLTP workloads).
- Internally, optimizer treats batch processing as new **physical property** (like sortedness) to combine operators in a proper way.

SQL Server: Performance

Performance impact (TPC-DS, scale factor 100, ≈ 100 GB):



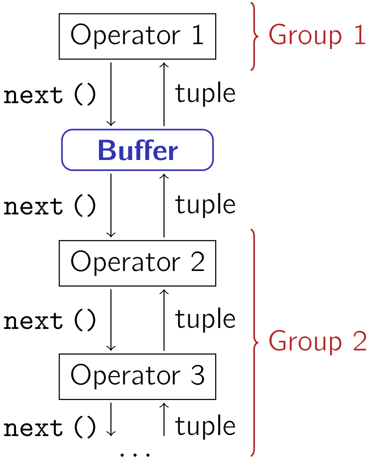
source: Larson *et al.* SQL Server Column Store Indexes. *SIGMOD 2011* (elapsed times, warm buffer pool).

Alternative: Buffer Operators

A similar effect can be achieved in a less invasive way by placing **buffer operators** in a pipelined execution plan.

- Organize query plan into **execution groups**.
- Add **buffer operator** between execution groups.
- Buffer operator provides tuple-at-a-time interface to the outside,
- but **batches up** tuples internally.

↗ Zhou and Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. *SIGMOD 2004*.

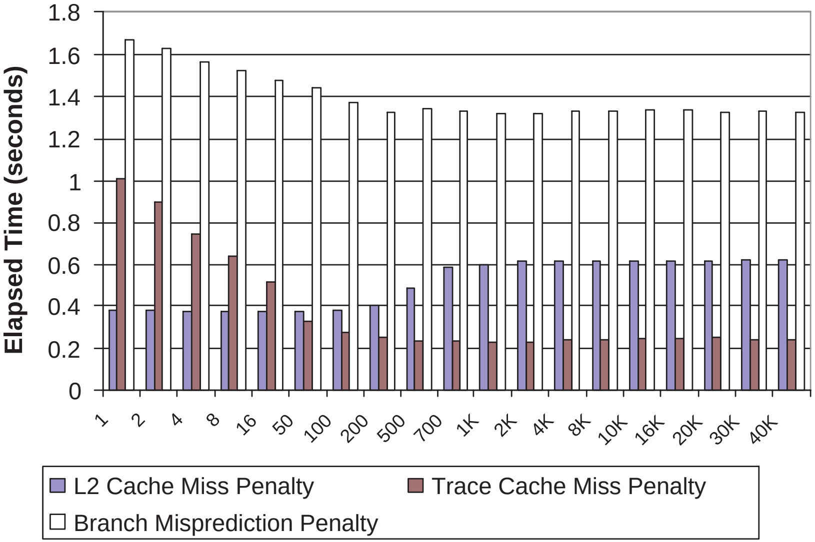


Buffer Operator

A buffer operator can be plugged into every Volcano-style engine.

```
Function: next ()
// Read a batch of input tuples if buffer is empty.
if empty and !end-of-tuples then
    while !full do
        append child.next () to buffer ;
        if end-of-tuples then
            break ;
// Return tuples from buffer
return next tuple in buffer ;
```

Buffer Operators in PostgreSQL



Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. *SIGMOD 2004.*

In-Memory Joins

After plain select queries, let us now look at **join queries**:

```
SELECT COUNT(*)
  FROM orders, lineitem
 WHERE o_orderkey = l_orderkey
```

(We want to ignore result construction for now, thus only **count** result tuples.)

We assume:

- no exploitable order,
- no exploitable indices (input might be an intermediate result), and
- an equality join predicate (as above).

Hash Join

Hash join is a good match for such a situation.

To compute $R \bowtie S$,

1. **Build a hash table** on the "outer" join relation R. — Build Phase
2. **Scan** the "inner" relation S and
   probe into the hash table for each tuple $s \in S$. — Join Phase

Function: `hash_join` (R, S)

// Build Phase

foreach tuple $r \in R$ **do**

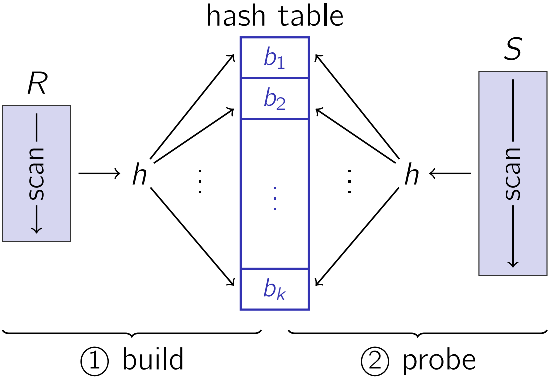
  insert s into hash table H ;

// Join Phase

foreach tuple $s \in S$ **do**

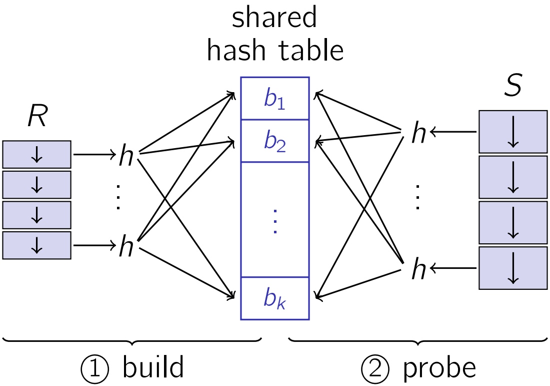
  probe H and append matching tuples to result ;

Hash Join



- ✓ $\mathcal{O}(N)$ (approx.)
- ✓ Easy to **parallelize**

Parallel Hash Join



✓ Protect using locks; **very low contention**

Modern Hardware

☹ **Random access pattern**

→ Every hash table access a **cache miss**

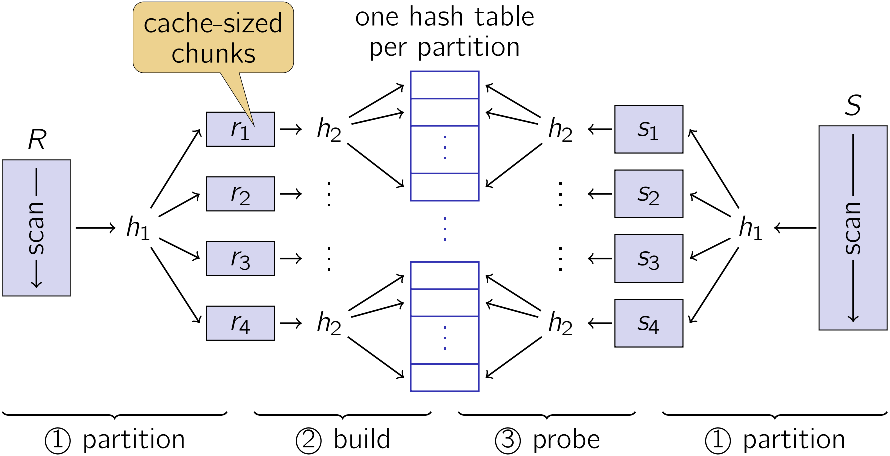
Cost per tuple (build phase):

- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

hash join
is severely
latency-bound

Partitioned Hash Join

Thus: **partitioned hash join** [Shatdal *et al.* 1994]



(parallelism: assign partitions to threads → no locking needed)

Cache Effects

Build/probe now contained within caches:

- 15/21 instructions per tuple (build/probe)
- ≈ 0.01 cache misses per tuple
- almost no TLB misses ✓

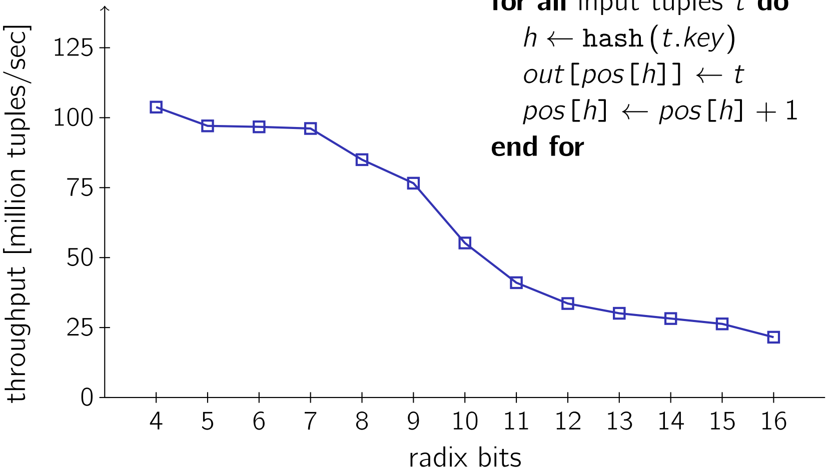
Partitioning is now critical

→ Many partitions, far apart

→ Each one will reside on its own page

→ Run out of **TLB entries** (100–500)

Cost of Partitioning



for all input tuples t **do**

$h \leftarrow \texttt{hash}(t.key)$

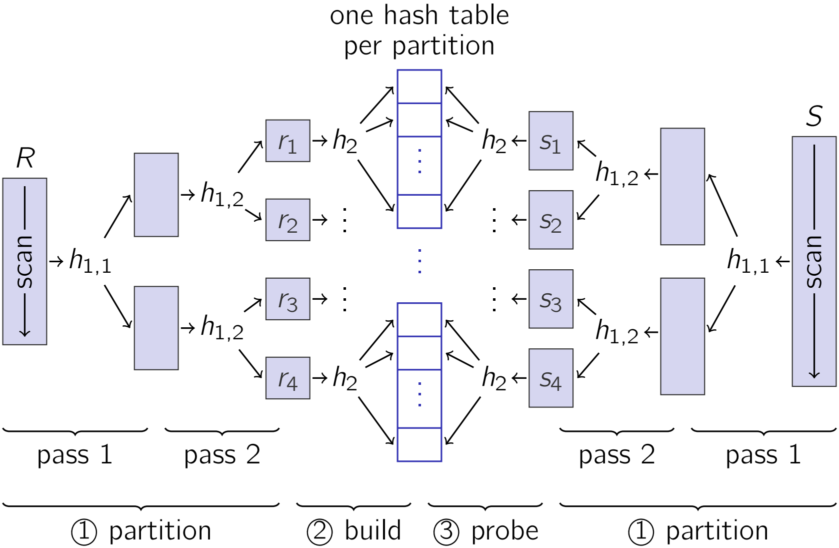
$out\texttt{[}pos\texttt{[}h\texttt{]]} \leftarrow t$

$pos\texttt{[}h\texttt{]} \leftarrow pos\texttt{[}h\texttt{]} + 1$

end for

→ Expensive beyond $\approx 2^8$–2^9 partitions.

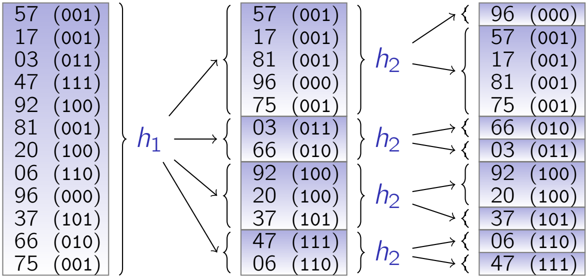
Multi-pass partitioning ("radix partitioning")



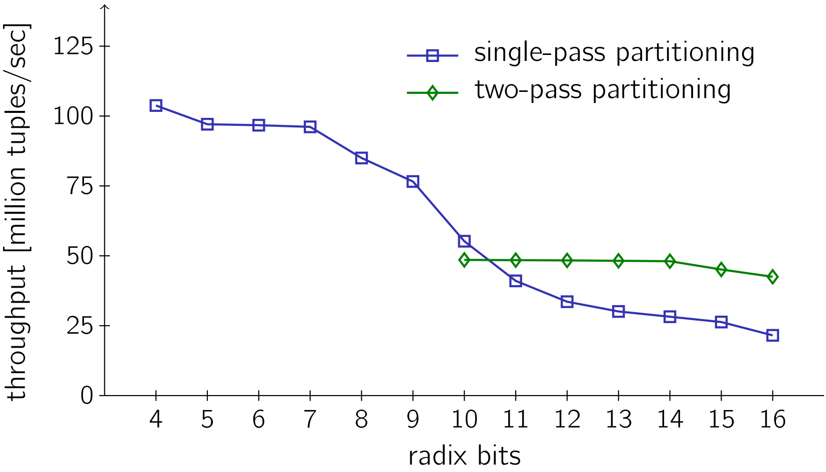
Multi-pass partitioning ("radix partitioning")

In practice:

- $h_1, \ldots, h_P$ use same hash function but look at different bits.



Two-pass partitioning



Hash join is $\mathcal{O}\big(N \log N\big)$!

for all input tuples t **do**
$h \leftarrow \texttt{hash}(t.key)$
copy t to $out[pos[h]]$
$pos[h] \leftarrow pos[h] + 1$
end for

memory access

Naïve partitioning (cf. slide 78)

for all input tuples t **do**
$h \leftarrow \texttt{hash}(t.key)$
$buf[h][pos[h] \;\texttt{mod}\; bufsiz] \leftarrow t$
if $pos[h] \;\texttt{mod}\; bufsiz = 0$ **then**
copy $buf[h]$ to $out[pos[h] - bufsiz]$
end if
$pos[h] \leftarrow pos[h] + 1$
end for

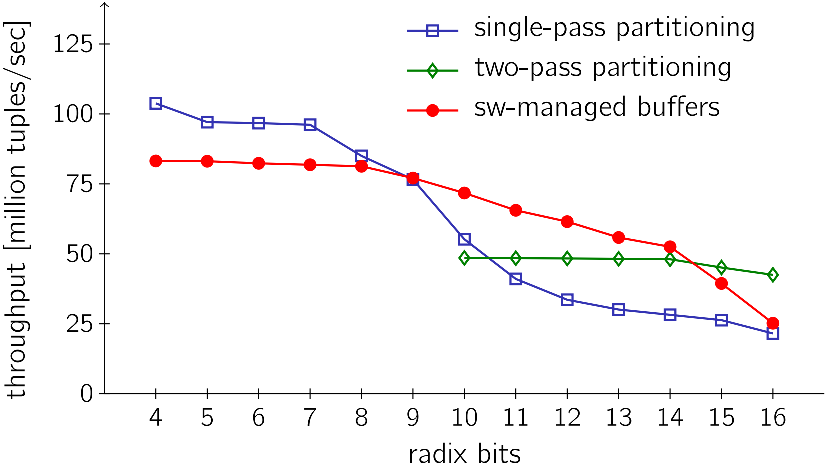
memory access

Software-Managed Buffers

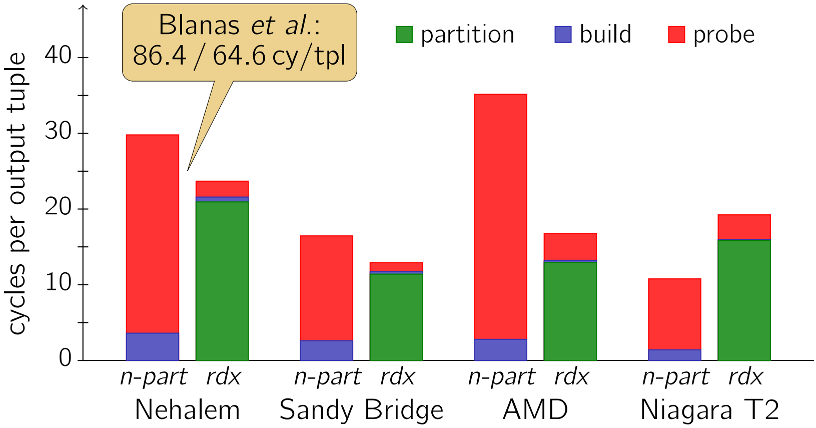
→ TLB miss only every *bufsiz* tuples

→ Choose *bufsiz* to match **cache line size**

Software-Managed Buffers



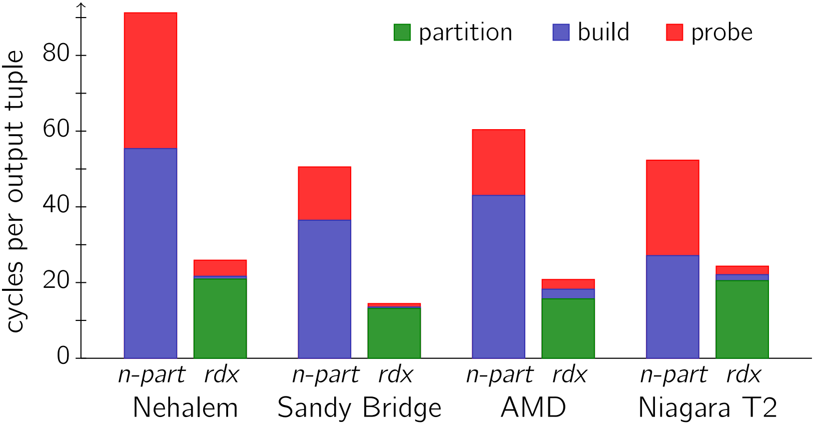
Plugging it together



- 256 MiB ⋈ 4096 MiB
- *e.g.*, Nehalem: 25 cy/tpl ≈ 90 million tuples per second

Nehalem: 4 cores/8 threads; 2.26 GHz · Sandy Bridge: 8 cores/16 threads; 2.7 GHz
AMD Bulldozer: 16 cores; 2.3 GHz · Niagara 2: 8 cores/64 threads; 1.2 GHz

Another Workload Configuration



- 977 MiB ⋈ 977 MiB
- *e.g.*, Nehalem: 25 cy/tpl ≈ 90 million tuples per second

Nehalem: 4 cores/8 threads; 2.26 GHz · Sandy Bridge: 8 cores/16 threads; 2.7 GHz
AMD Bulldozer: 16 cores; 2.3 GHz · Niagara 2: 8 cores/64 threads; 1.2 GHz

Resulting Overall Performance

Overall performance is influenced by a number of parameters:

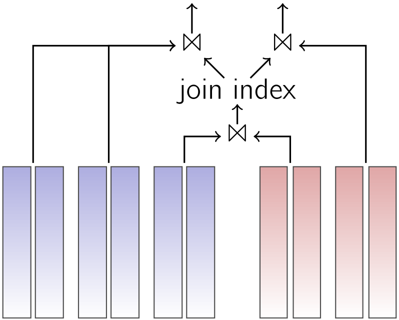
- input data volume
- cluster size / number of clusters
- number of passes (plus number of radix bits per pass)

An **optimizer** has to make the right decisions at runtime.

- Need a detailed **cost model** for this.

Joins and Column-Based Storage

With column-based storage, a single join is not enough.



- Joining BATs for key attributes yields a **join index**.
- **Post-project** BATs for all remaining attributes.

Joins and Column-Based Storage

Positional lookup?

- Makes post-projection joins "random access" ☹

Thus:

- **(Radix-)Sort** by `oid`s of larger relation
  → Positional lookups become cache-efficient.
- **Partially cluster** by `oid`s before positional join of smaller relation
  → Access to smaller relation becomes cache-efficient, too.

Details: Manegold, Boncz, Nes, Kersten. Cache-Conscious Radix-Decluster Projections. *VLDB 2004*.