Main Memory DBMS

G. Moerkotte

February 4, 2019

Contents

Chapter 1

Introduction

The holy grail for a DBMS is one that is [183]:

- Scalable & Speedy,
  to run on anything from small ARM processors up to globally distributed compute clusters,
- Stable & Secure,
  to service a broad user community,
- Small & Simple,
  to be comprehensible to a small team of programmers,
- Self-managing,
  to let it run out-of-the-box without hassle.

We will have a more limited view, shared with Stonebraker:

> There are three important things in databases:
>
> 1. performance,
> 2. performance, and
> 3. performance.

We concentrate on the architecture and the implementation of Query Execution Engines (QEE) focussing on *data layout* and (algebraic, simple) *expression evaluation*.

We start out with a short refresh on hardware and a reminder that operating systems are our friends. Then, we recall some stuff on hash tables and drop the names of light-weight compression techniques that are applied in database systems.

The first true chapter is Chap. 6. The chapter on memory management is quite short yet. Larger is the chapter on the principal workings of a physical algebra (Chap. 7). Very large is (or will be) the chapter on expression evaluation (Chap. 8).

OTHER CHAPTERS TODO

Chapter 2

Hardware

2.1 Memory

2.1.1 Aligned vs. Unaligned Access

Accessing a data item d at memory address a is aligned if

$$a \mod |d| = 0$$

if $|d|$ is the size of the data item in bytes.

Fig. 2.1 illustrates aligned vs. unaligned access of a 4-byte integer. Unaligned access maybe more expensive or even impossible on some architectures.

2.1.2 Address Translation (Virtual Memory)

Programs work with virtual addresses that must be translated into physical addresses. An abstract view of this mapping is shown in Fig. 2.2.

Since mapping single addresses is way to expensive, the virtual and the physical address space is organized into *pages* of equal size. A typical page size is 4 KB, but some processors support several/other page sizes. Then, the mapping is from a page of the virtual address space to a page in the physical space. The offset within the virtual address space remains the same in the physical address space. This is illustrated in Fig. 2.3.

At the bit-level, the address translation is illustrated in Fig. 2.4. The offsets bits from the virtual address are copied to the according bits in the physical address and just the logical page number is translated into the physical page number. Typically/currently, not all bits of a 64-bit logical address are used. [some researchers even make use of these unused bits [169]]

The mapping from virtual page numbers to physical page numbers is implemented via a *page table*, which contains for every possible virtual page number the according physical page number. To find the beginning of the page table, a register called *translation table base register* (TTBR) is used. The implementation of the mapping is illustrated in Fig. 2.5.

Since the page table is kept in main memory, translating a virtual address to a physical address would require an additional memory access. Obviously,

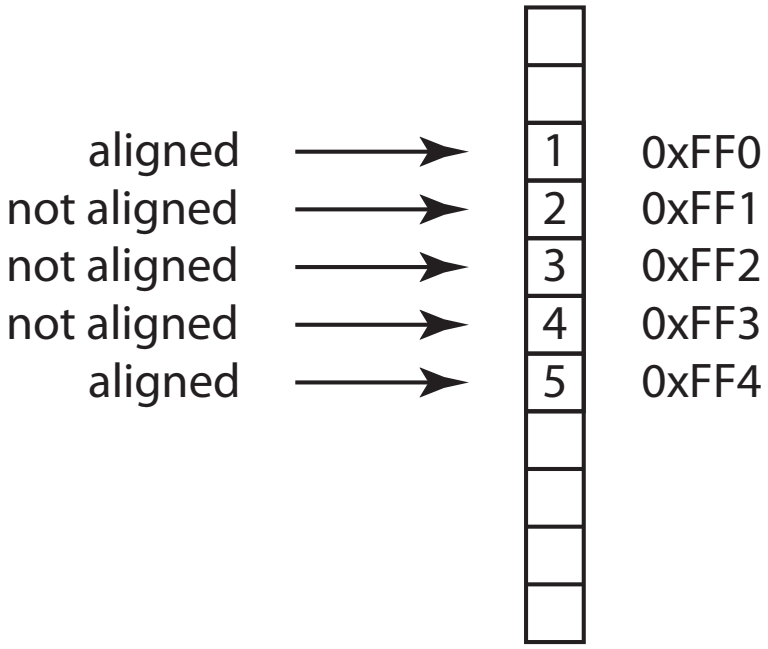


Figure 2.1: Aligned vs. unaligned access

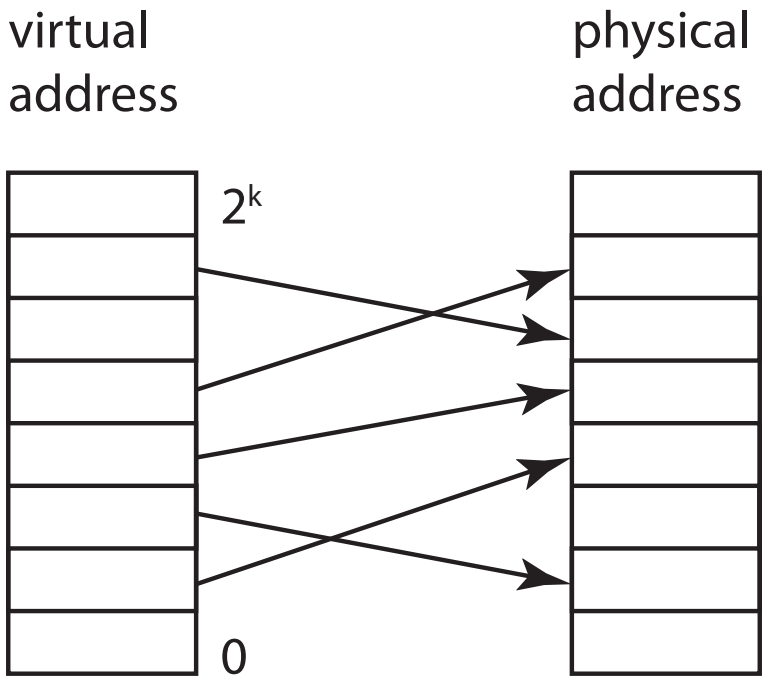


Figure 2.2: Virtual address to physical address mapping: schema

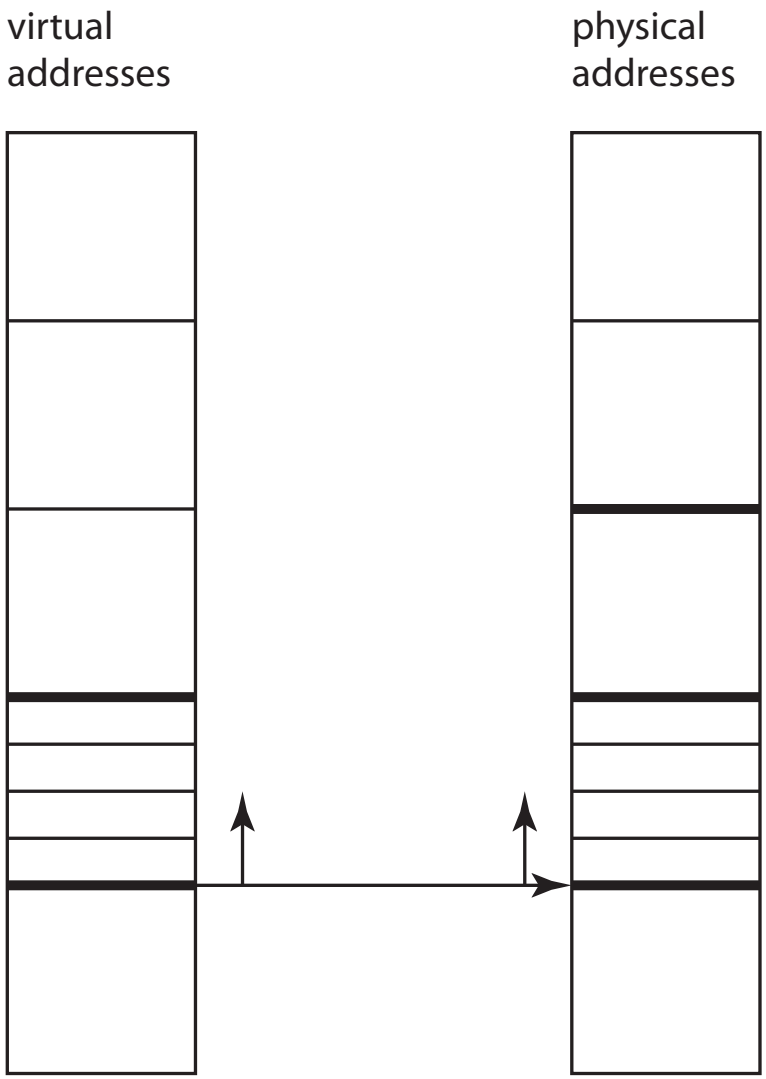


Figure 2.3: Virtual address to physical address mapping: pages

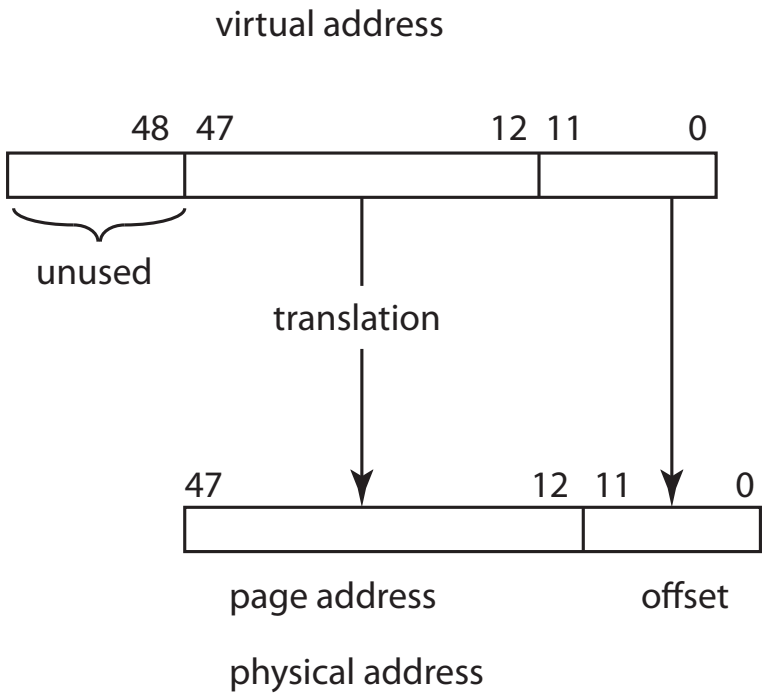


Figure 2.4: Virtual address to physical address mapping: page+offset

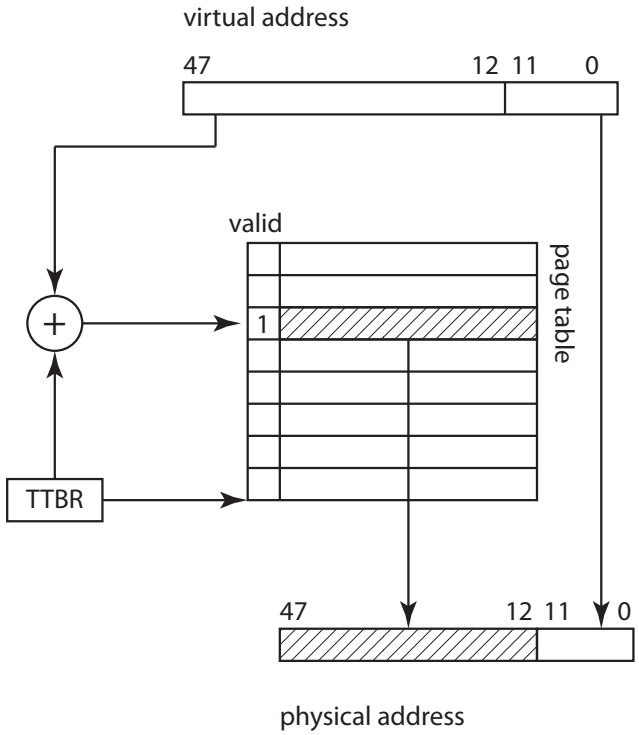


Figure 2.5: Virtual address to physical address mapping: page table

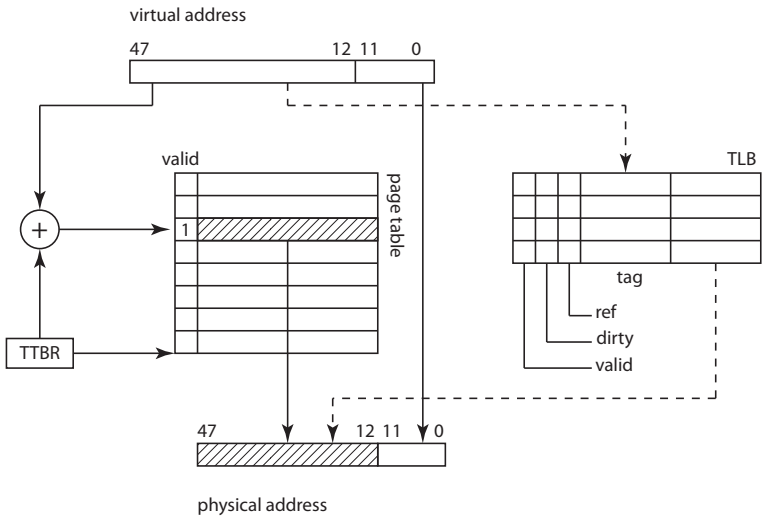


Figure 2.6: Virtual address to physical address mapping: TLB

this is very expensive. Thus, a piece of special memory called *TLB* (translation lookaside buffer) is introduced to cache part of the translation table (see Fig. 2.6).

Typically, there exist TLB1 and TLB2 caches for the translation table. Example Intel i7-4790:

- instruction TLB1 [for 4KB pages]: 64 entries, 8-way
- data TLB1 [for 4KB pages]: 64 entries, 4-way
- TLB2 cache [for 4KB pages]: 1024 entries, 8-way

Exercise:

1. find out the page size of your computer
2. find out the number of TLB1/2 entries of your computers as well as their latencies.
3. Assume a page size of 4KB and assume there are 64 TLB1 entries. How large is the address space accessable without TLB1 misses.

2.1.3 Caches and the Memory Hierachie

Features of caches:

- Quantitative features:
  - size
  - associativity
  - hierarchy

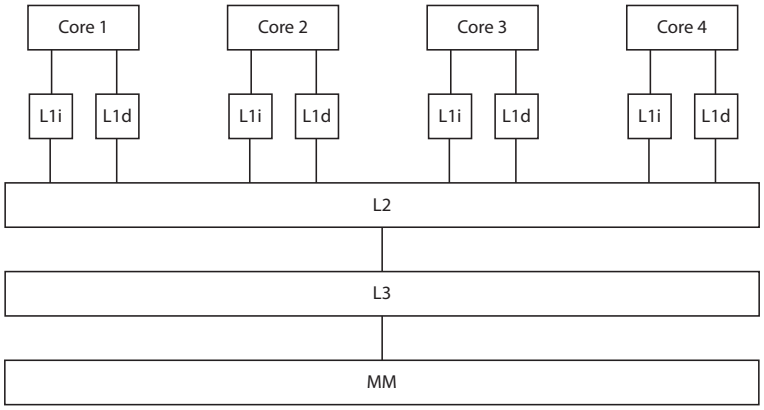


Figure 2.7: One possible organization of caches (L1 per core, shared L2, (optional) shared L3)

  - latency

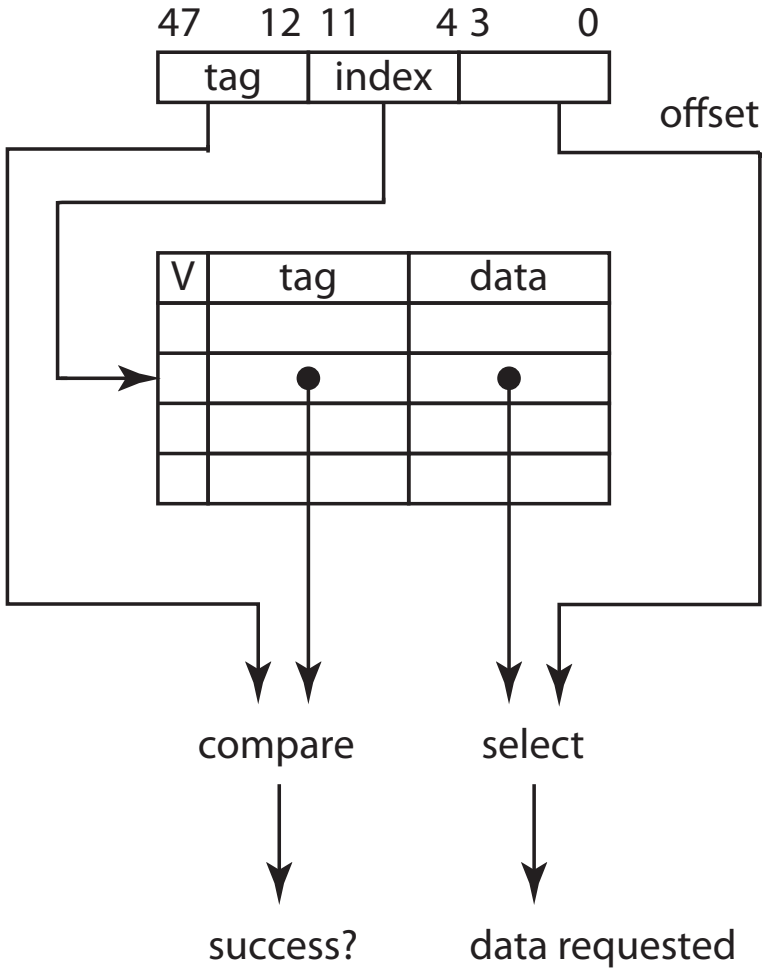
- Qualitative features:

  - nonblocking caches [cache can serve accesses while processing a miss]
  - way prediction [predicts way of the next access to safe comparisons]
  - victim caches [cache holds evicted cache lines]
  - trace caches [L1i]
  - can cache on virtual or physical addresses
  - inclusive/exclusive

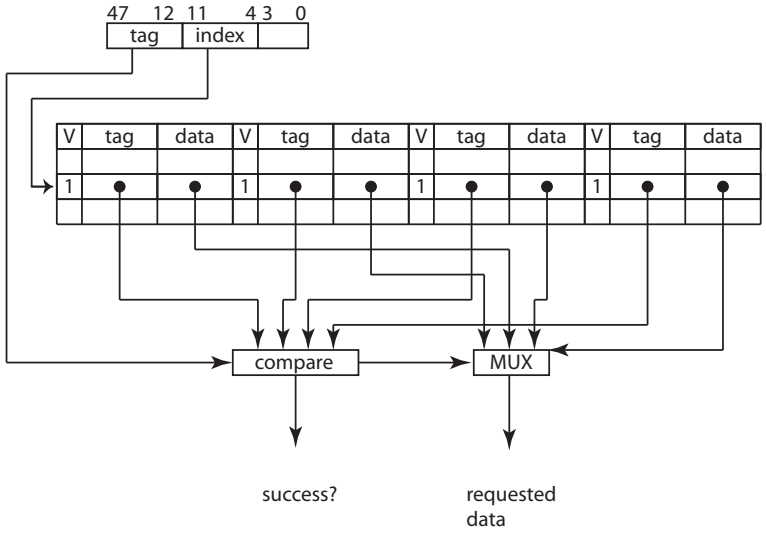
Example organization of the memory hierarchy (see Fig 2.7).
Latencies (approximate):

mem	latency [cycles]
register	≤ 1
L1	3-4
L2	≈ 14
TLB1	≈ 12
TLB2	≈ 30
main memory	≈ 240

Detailed organization of a cache, non-associative:



Detailed organization of a cache, 4-way associative:



Exercise:

1. Measure the cache latencies of your computer

CPU	L1i	L1d	L2	L3	L4	L1 TLB entries
Power8	32 KB	64 KB	512 KB/core	8 MB/core	16 MB/bc	72i + 48d
Xeon E5 v4	32 KB	32 KB	512 KB	2.5 MB/core		128i + 64d
i7-4790	32 KB	32 KB	256 KB	8 MB		64i + 64d
Exynos 2254	32 KB	32 KB	2 MB	-		32i + 32d ?

Table 2.1: Cache sizes for some architectures

2. Describe what can happen if we write a 4-byte integer to some memory address belonging to a cache line not present in any cache.

2.1.4 Some Numbers of Some Processors

Table 2.1 contains the sizes of caches for some architectures.

Remarks:

- Exynos 5422: 4x Cortex A15 share 2 MB L2-cache, 4x A7 share 512 KB L2-cache.
- 4-K pages
- Power8, Xeon: L1: 8-way associative
- Power8: L3 shared, 8 MB per core, 12 core, 96 MB, 8-way associative; L4: shared, 16-way associative; up to 8 buffer chips (bc)
- A15: 2-way associative
- Power8: 128 B cache lines, 2048 entries in TLB
- Intel i7-4790: 64 B cache lines, L1i/d, L2: 8-way associative, L3: 16-way associative
- sources: anandtech, hardkernel, 7-cpu, Power8 Performance Optimization and Tuning Techniques, and [239, 243]

Exercise:

1. Add the new Ryzen 7 1800+ processor to Table 2.1

2.1.5 Prefetching

Hardware prefetcher:

- adjacent cache line prefetcher
- stride prefetcher

Often: prefetchers do not prefetch across page boundaries.

Software prefetching:

- explicit prefetch instructions

Performance of hardware prefetcher for sequential access. Measure code fragment:

```
1  for (int i = 0; i < n; ++i)
2          r += A[I[i]]
```

I index array filled in two different ways:

1. contains consecutive numbers $[0, n[$
2. contains random permutation of $[0, n[$

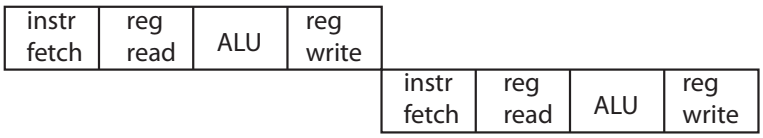
results in time per element:

kind of read	i7-4790	i7-4790	Exynos 2254
n	10^9	10^8	10^8
random	45.7 ns	11.3 ns	43.6 ns
sequential	0.8 ns	0.8 ns	3.2 ns
factor	57.1	14.1	13.6

2.2 CPU

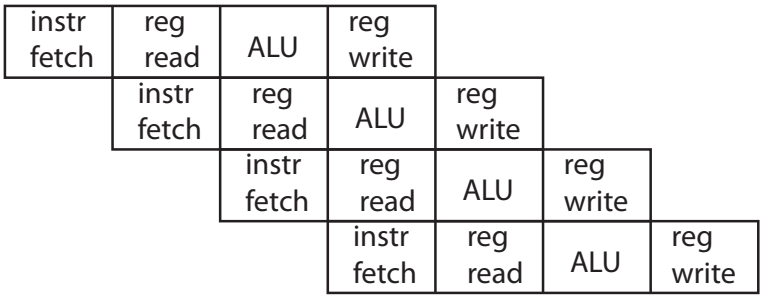
2.2.1 Pipelining

Illustration of non-pipelined execution (simplified):



We get an IPC of 0.25.

Illustration of pipelined execution:



We get an IPC of 1.

Pipeline hazards (resulting in stall):

- data dependency
- data access
- branch misprediction
- instruction stall

Worst of all is the *instruction stall*, where the core has (due to memory/cache access latencies) no instruction to execute.

2.2.2 Out-Of-Order-Execution

- ops $\rightarrow$ μ-ops
- μ-ops processed at ports (concurrently, obeying data dependencies)
- μ-ops sometimes in caches (saves instruction decode)

Example (Haswell) [122]:

Port 0 : integer ALU and shift, FMA FP mult, vector int multiply, vector logicals, branch, devide, vector shifts

Port 1 : integer ALU and shift, FMA FP mult and FP add, vector int ALU, vector logicals

Port 2 : load and store address

Port 3 : load and store address

Port 4 : store data

Port 5 : integer ALU and LEA, vector shuffle, Vector int ALU, vector logicals

Port 6 : integer ALU and shift

Port 7 : store address

Example (Power8) [239] (16 exeuction pipelines per core):

- 2 fixed-point
- 2 load/store
- 2 load
- 4 double (can act as 8 float)
- 2 VMX/VSX
- 1 crypto
- 1 branch execution
- 1 condition register logical
- 1 decimal floating-point

To illustrate the out-of-order processing for `read` operations, i.e., parallelizing memory accesses, we repeat an experiment performed by Manegold, Boncz, and Kersten [181]. We sum up all elements in an array containg $n = 10^8$ elements using two different functions. The first one is the simple, standard implementation:

```
int
sum0(int* arr, int n) {
    int lSum = 0;
    for(int i = 0; i < n; ++i) {
        lSum += arr[i];
    }
    return lSum;
}
```

The second one uses two partitial sums, one for each half of the array:

```
int
sum1(int* arr, int n) {
    int lHalf = n/2;
    int lSum = 0, lSum1 = 0, lSum2 = 0;
    for(int i = 0; i < lHalf; ++i) {
        lSum1 += arr[i];
        lSum2 += arr[i+lHalf];
    }
    lSum = lSum1 + lSum2;
    if(n & 0x1) {
        lSum += arr[n-1];
    }
    return lSum;
}
```

The execution times per element in the array on a i7-4790 are:

sum0	0.375 ns
sum1	0.254 ns

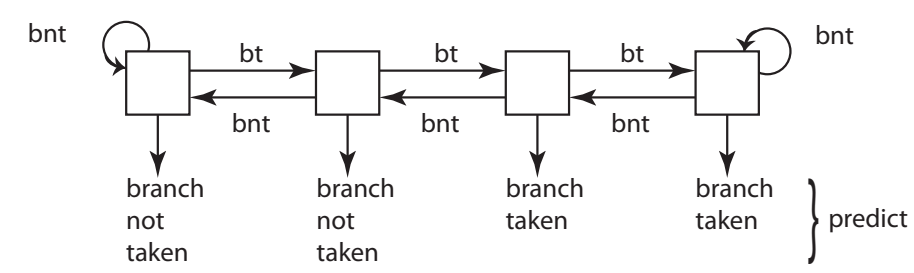
where we compiled with `gcc -O2`.

Exercise:

1. Compile the functions `sum0` and `sum1` with `gcc -O2` and `gcc -O3`. Measure the execution times and analyze at the assembler code produced.

2.2.3 (Cost of) Branch (Mis-) Prediction

Schematic 2-bit branch predictor:



Code with branch

code fragment 1: code with jump

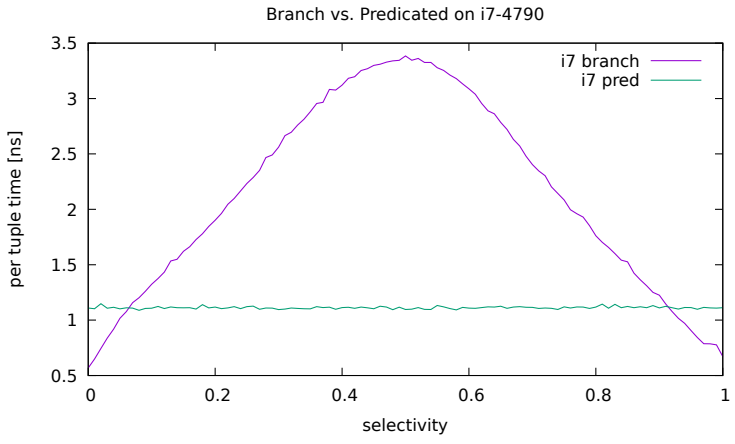
```
SELECT(int* b, int* a, int l, int n)
1  int j = 0;
2  for (int i = 0; i < n; ++i)
3        if (a[i] < l)
4              b[j++] = a[i];
5  return j
```

Predicated Code

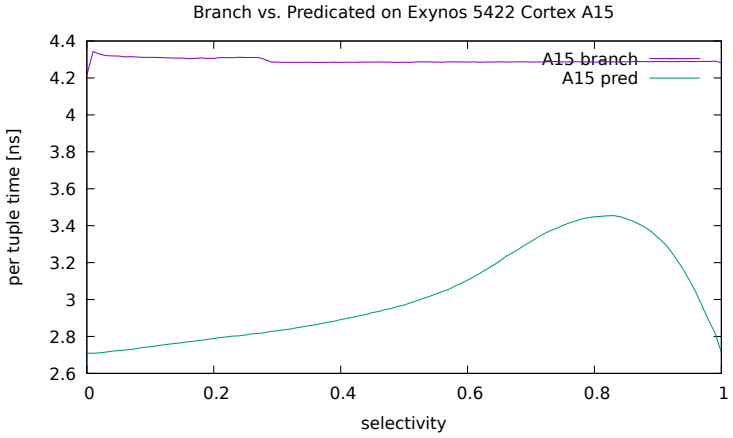
code fragment 2: predicated code [225, 226]

```
SELECT(int* b, int* a, int l, int n)
1  int j = 0;
2  for (int i = 0; i < n; ++i)
3        b[j] = a[i];
4        j += (a[i] < l)
5  return j
```

Experiment with i7-4790



Experiment with Samsung Exynos 6422 Cortex A15



[Note: conditional execution of instructions on ARM]

Exercise:

1. use `godbolt` to study the assembler code generated by different compilers for different architectures for the above two code fragments.
2. Discuss the code fragment below where AND is defined as one of
   - #define AND &
   - #define AND &&

Code fragment for exercise:

```
int
select_between(b, a, n, lb, ub)
    j = 0;
    for(i = 0; i < n; ++i)
        if((lb[i] ≤ a[i]) AND (a[i] ≤ ub[i]))
            b[j++] = a[i];
```

2.2.4 SIMD

Idea:

- perform the same operation on multiple operands at the same time
  SIMD = single instruction multiple data

Supported by virtually all processors:

- ARM: NEON
- Intel: SSE, AVX
- Power: VMX, VSX

- Sparc: VIS

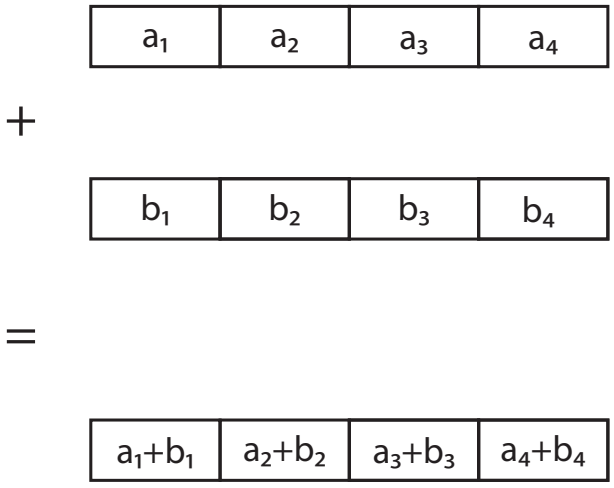
Usable

- automatically by compiler
- manually (inline assembler/intrinsics)

There are hundreds if not thousands SIMD instructions. In the current lecture, we briefly look at some SIMD instructions provided by some Intel CPUs but skipping most of those given below.

Arithmetics

Example illustration of a SIMD `add` operation:



Example instructions (Intel):

- __m256i _mm256_add_epi32(__m256i a, __m256i b)
- __m256i _mm256_sub_epi32(__m256i a, __m256i b)
- __m128i _mm_mullo_epi32(__m128i a, __m128i b)
- __m256i _mm256_udiv_epi32(__m256i a, __m256i b)
- __m256i _mm256_urem_epi32(__m256i a, __m256i b)
- __m256i _mm256_udivrem_epi32(__m256i * mem_addr, __m256i a, __m256i b)

Latencies vary between architectures. see

- software.intel.com/sites/landingpage/IntrinsicsGuide

bit operations

- __m256i _mm256_and_si256(__m256i a, __m256i b)
- __m256i _mm256_andnot_si256(__m256i a, __m256i b)
- __m256i _mm256_or_si256(__m256i a, __m256i b)
- __m256i _mm256_xor_si256(__m256i a, __m256i b)

plus several operations to test bits

shift

arithmetic/logical shift left/right, e.g.,

1. __m256i _mm256_slli_epi32 (__m256i a, int imm8)
2. __m256i _mm256_sllv_epi32 (__m256i a, __m256i count)

load

1. __m256i _mm256_lddqu_si256 (__m256i const* mem_addr)
   unaligned
2. __m256i _mm256_load_si256 (__m256i const * mem_addr)
   32-byte aligned
3. __m256i _mm256_stream_load_si256 (__m256i const* mem_addr)
   32-byte aligned

Effect:

dst[255:0] := MEM[mem_addr+255:mem_addr]

broadcast load

load a single number into all elements of a SIMD-register:

- __m128 _mm_load1_ps (float const* mem_addr)
  Load a single-precision (32-bit) floating-point element from memory into all 4 elements of dst.
- __m256 _mm256_broadcast_ss (float const* mem_addr)

selective load

- __m256i _mm256_maskload_epi32 (int const* mem_addr, __m256i mask)

```
FOR j := 0 to 7
    i := j*32
    IF mask[i+31]
        dst[i+31:i] := MEM[mem_addr+i+31:mem_addr+i]
    ELSE
        dst[i+31:i] := 0
    FI
ENDFOR
```

partial load

- __m128i _mm_loadl_epi64 (__m128i const* mem_addr)

effect:

```
dst[63:0] := MEM[mem_addr+63:mem_addr]
dst[MAX:64] := 0
```

gather load

- __m256i _mm256_i32gather_epi32 (int const* base_addr, __m256i vindex, const int scale)

```
FOR j := 0 to 7
    i := j*32
    dst[i+31:i] := MEM[base_addr + SignExtend(vindex[i+31:i])*scale]
ENDFOR
```

store

- void _mm256_store_si256(__m256i* mem_addr, __m256i a)
  32-byte aligned

Effect:

```
MEM[mem_addr+255:mem_addr] := a[255:0]
```

Exercise:

1. Write a simple program using SIMD that adds up all elements in an array.

selective store

- void _mm256_maskstore_epi32 (int* mem_addr, __m256i mask, __m256i a)

```
FOR j := 0 to 7
    i := j*32
    IF mask[i+31]
```

```
        MEM[mem_addr+i+31:mem_addr+i] := a[i+31:i]
    FI
ENDFOR
```

scatter store

```
void _mm256_mask_i32scatter_epi32(void* base_addr,
                                  __mmask8 k,
                                  __m256i vindex,
                                  __m256i a,
                                  const int scale)
```

effect

```
FOR j := 0 to 7
    i := j*32
    IF k[j]
        MEM[base_addr + SignExtend(vindex[i+31:i])*scale] := a[i+31:i]
        k[j] := 0
    FI
ENDFOR
```

Useful is the detection of conflicts (writes to the same location):

- _mm256_conflict_epi32
- _mm512_conflict_epi32

Test each 32-bit element of r for equality with all other elements in r closer to the least significant bit. Each element's comparison forms a zero extended bit vector in dst:

```
FOR j := 0 to 7
    i := j*32
    FOR k := 0 to j-1
        m := k*32
        dst[i+k] := (a[i+31:i] == a[m+31:m]) ? 1 : 0
    ENDFOR
    dst[i+31:i+j] := 0
ENDFOR
dst[MAX:256] := 0
```

compare

- __m256i _mm256_cmpeq_epi32(__m256i a, __m256i b)
- __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

effect of 1:

```
FOR j := 0 to 7
    i := j*32
    dst[i+31:i] := ( a[i+31:i] == b[i+31:i] ) ? 0xFFFFFFFF : 0
ENDFOR
```

collecting compare results into bitvector

Set each bit of mask dst to the most significant bit of the 32-bit element in a.

- int _mm256_movemask_ps(__m256 a)

effect:

```
FOR j := 0 to 7
    i := j*32
    IF a[i+31]
        dst[j] := 1
    ELSE
        dst[j] := 0
    FI
ENDFOR
```

convert

- __m256i _mm256_cvtepi8_epi32 (__m128i a)

effect:

```
FOR j := 0 to 7
    i := 32*j
    k := 8*j
    dst[i+31:i] := SignExtend(a[k+7:k])
ENDFOR
```

blend

- __m256i _mm256_blend_epi32(__m256i a, __m256i b, const int imm8)

effect:

```
FOR j := 0 to 7
    i := j*32
    IF imm8[j%8]
        dst[i+31:i] := b[i+31:i]
    ELSE
        dst[i+31:i] := a[i+31:i]
    FI
ENDFOR
```

shuffle/permute

- __m256i _mm256_permutevar8x32_epi32 (__m256i a, __m256i idx)

effect:

```
FOR j := 0 to 7
   i := j*32
   id := idx[i+2:i]*32
   dst[i+31:i] := a[id+31:id]
ENDFOR
```

Bit Manipulations

- `pop_count`
- `_bit_scan_forward`, `_bit_scan_reverse`
- `_pdep_u32`, `_pext_u32`

pop-count does not exist for large SIMD registers. Fast implememtations are provided by [194].

other useful instructions accessible by builtins are:

`blsr`(a)	:=	$a \oslash (a-1)$	// reset lowest bit set
`blsi`(a)	:=	$a \oslash (-a)$	// extract lowest bit set
`blsmsk`(a)	:=	$a \otimes (a-1)$	// set all lower bits up to incl. lowest bit set
`tzcnt`(a)			// count number of trailing zero bits
`lzcnt`(a)			// count number of leading zero bits

Exercise:

1. implement some methods presented in [194] for NEON registers.

Prefetching

Sometimes it is beneficial to use explicit prefetching instructions to hide memory access latencies. There is a useful built-in to support this:

- __builtin_prefetch(void* mem, int rw, int a)

where

mem is the memory address to be prefetched

rw indicates prefetching for read (0) or write (1)

a indicates the access pattern: $a = 0$ indicates that the temporal locality is low, that is, we probably don't access the data item again after the first access. $a = 3$ indicates the contrary, $a = 2$ something inbetween

Streaming Stores

Bypass cache by streaming stores. Instructions, e.g.:

- _mm256_stream_si256
- _mm512_storenrngo_ps
- _mm512_storenrngo_pd

The latter two also follow a weaker memory model.

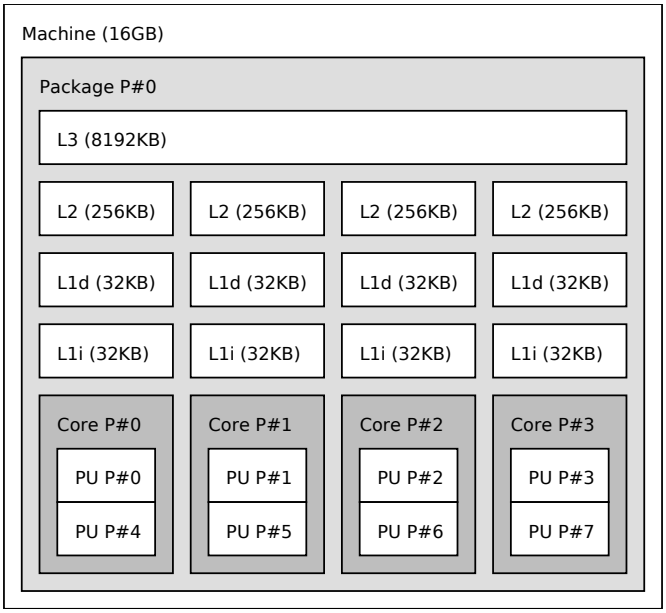
When writing to two different locations within a single cache line, it may happen that the cache line is written twice to main memory. To prevent this, some processors provide *write-combine* buffers, which combine multiple writes to a cache line in order to write it only once to main memory.

Software can make use of it by issueing two streaming store operations (e.g. `_mm256_stream_si256`) in close neighborhood which together cover a whole cache line. This is called *software write-combining*.

2.2.5 Simultaneous multithreading (SMT)

- AMD/Intel: 2 threads per core
- Power8: up to 8 threads per core

Example (produced by lstopo):



2.3 Cache Coherence

Cache coherence makes sure that simultaneous memory accesses to the same cache line by different cores does not result in any correctness problems [185,

242]. (As long as the memory addresses are not the same!). However, this may lead to performance problems as illustrated below.

The most commonly used basic protocol is the MESI protocol where each cache line can be in one of four states:

modified the cache line has been modified
no other processor has this cache line

exclusive the cache line has not been modified
no other processor has this cache line

shared the cache line has not been modified
other processors may have this cache line

invalid the cache line does not hold any valid data

The MESI protocol than works around these states ensuring their guarantees. For details about the MESI protocol see [185]. Intel uses the MESIF protocol, with an additional *Forward* state.
Function code incrementing a pointer on a given hw-thread:

```
void
f(uint64_t* s, uint64_t n, int aHwThreadNo) {
    cbind_to_hw_thread(aHwThreadNo, 1);
    for(uint64_t i = 0; i < n; ++i) {
        *s += 1;
    }
}
```

Two threads with this functions are run simultaneously with different hw-thread numbers and different pointers. In one case, both pointers point into the same cache line, in the other case, they point into different cache lines. The following table contains the runtime results (both threads finish, $n = 10^9$):

	HW thread no		exec time for pointer distance	
CPU	HWT 1	HWT 2	8 B	800 B
Intel i7-4790	4	7	5.37 s	1.52 s
	3	7	3.33 s	2.22 s
Exynos 5422	4	7	4.75 s	4.89 s

Recall: Intel i7-4790 supports SMT: hw-threads $[0, 4]$ are on core 0, $[1, 5]$ are on core 1, $[2, 6]$ on core 2, $[3, 7]$ on core 3.

Exercise:

1. Find out why there is virtually no performance penalty for the Exynos 5422.

2.4 Synchronization Primitives

In order to synchronize different threads and prevent race conditions, synchronization primitives such as `mutex` and `semaphore` must be used and implemented. This is facilitated by atomic operations provided by the underlying hardware. Typical operations implementing atomicity are:

- compare-and-swap (CAS)
- fetch-and-add (FAA)
- load exclusive, store exclusive (LDREX, STREX on ARM)
- memory barrier instructions (e.g. `DMB` on ARM, `mfence` on Intel)

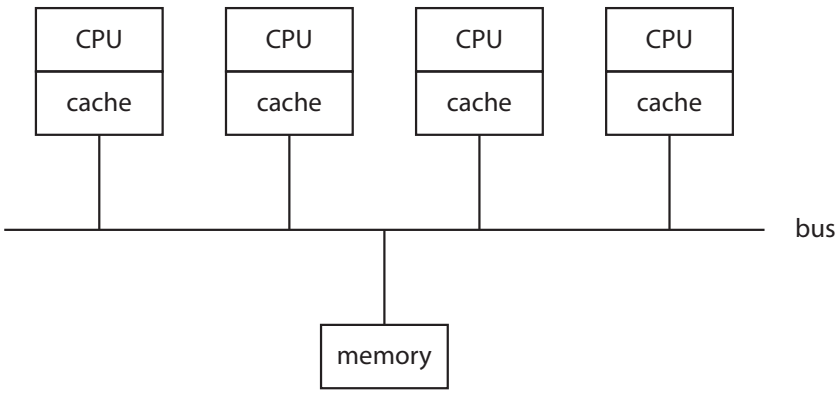
More on this can be found in [20, 77, 119, 133, 185, 242].

2.5 NUMA

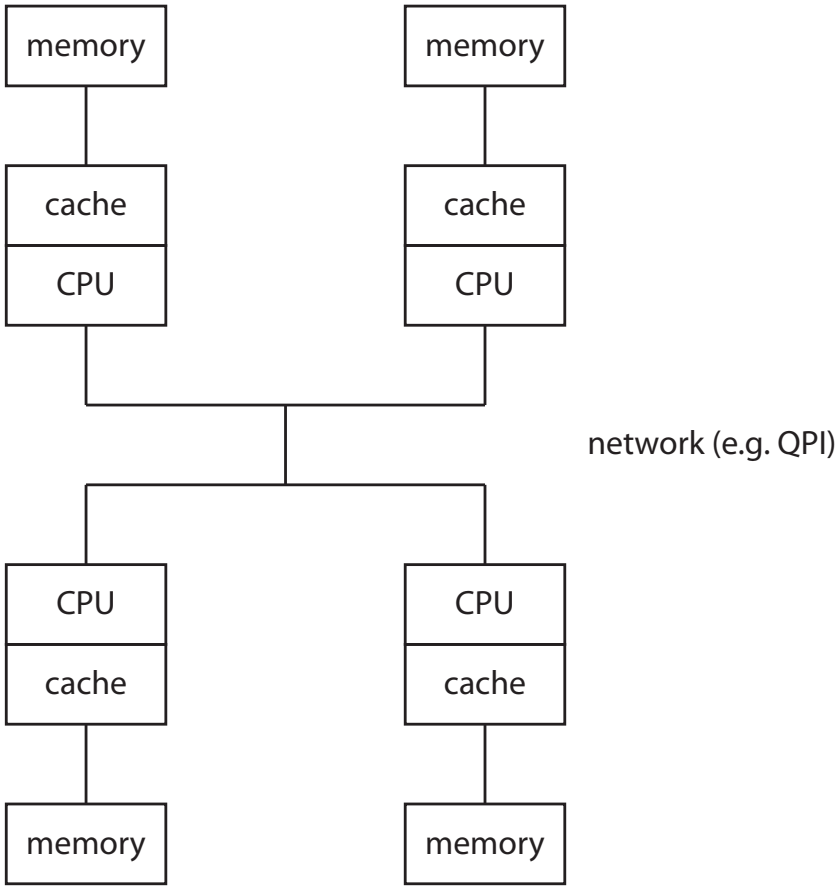
old from all CPU cores access to memory via a single bus (UMA/SMP)

new non-uniform memory access (NUMA)

Sketch of UMA:



Sketch of NUMA:



Example (produced by lstopo):

Some measurements by IntelMemoryLatencyChecker for a Xeon E5-2690 v3 @ 2.60GHz:

Measuring idle latencies [ns]		
	Numa node	
Numa node	0	1
0	78.8	123.5
1	122.5	79.4

Memory Bandwidths [MB/s]		
	Numa node	
Numa node	0	1
0	61112.6	18817.4
1	18942.2	61207.7

Exercise:

1. Write a simple C/C++ program investigating the topology of the computer it runs on. (like lstopo but only textual output)

2. Find out the name of the interconnect used for AMD Naples.

2.6 Performance Monitoring Unit

Processors have (configurable) hardware performance counters for different events:

- cycle/instruction counter
- caches/memory: read access, write access, refill

Example: ARMv7

- use coprocessor registers
- 1 non-configurable cycle counter
- 6 configurable counters

2.7 References

- Patterson, Hennessy: Computer Organization and Design; The Hardware/Software Interface; ARM Edition; 2016.
- Hennessy, Patterson: Computer Architecture; A Quantitative Approach. 2016.
- Drepper: What Every Programmer Should Know About Memory. 2007. [article, online]

Chapter 3

Operating System

useful system calls for

1. process/thread support, binding threads to cores
2. cooperation
    - shared memory
3. communication
    - ipc
    - network
4. I/O
    - raw I/O
    - direct I/O
    - chained I/O
    - vectorized I/O
    - memory mapped files
5. numa
6. fork (copy on write) application in Hyper: [98]
7. clock access
8. hardware inspection

Chapter 4

Hash Tables

4.1 Hash Functions

The following list tells us that hash functions are an important and well-studied problem:

- division

  $$h(x) := x \mod p$$

  or, in general,

  $$h_{a,b}(x) := ((ax + b) \mod p) \mod n$$

  where $a, b \in \mathbb{Z}_p$.

- multiplicative
  $h(x) := \lfloor m(\frac{a}{w}x \mod 1) \rfloor$

- fibonacci hashing

- polynomial over prime field (k-universal)
  $h(x) := \sum_{i=0}^{k-1} a_i x^i \mod p$

- multiply-(add)-shift (2-universal)
  $h_{a,b}(x) := (ax + b) \gg (l - l_{\text{out}})$
  or plain multiplicative:
  $h_a(x) := (ax) \gg (l - l_{\text{out}})$

- murmur

- tabulation, $x = [a_0, \ldots, a_{q-1}]$
  $h(x) := h_0[a_0] \oplus \ldots \oplus h_{q-1}[a_{q-1}]$

- hashpjw

- CityHash

- Fowler-Noll-Vo, Jenkins, SpookyHash, Zobrist

- Larson
  while(*s) h = h * 101 + *s++

references

- Knuth: The Art Of Programming Vol 3 [153].
- Aho, Sethi, Ullman: Compilers/Compilerbau.
- Carter, Wegman: Universal hashing [55]
- Ramakrishna, Bahcepkapili: Efficient Hardware Hashing Functions for High Performance Computers. IEEE Trans. on Computers, 46(12), 1997.
- Ramakrishna, Zobel: Performance of String Hashing Functions [217]
- Patrascu, Thorup: The Power of Simple Tabulation Hashing [210]
- Thorup, Zhang: Tabulation-based 4/5-universal hashing [249, 250]
- Dietzfelbinger et al: Multiplicative Universal Hashing [84]
- Richter et al: A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. VLDB 2015.
- Ramakrishna, Zobel: hashing functions for strings, introduce add-shift-xor function [217]
- Gonnet: expected length of longest collision chain [103]

4.1.1 Why hash-functions matter

Simple experiment: encode dates from 01.01.1950 to 31.12.1999 into 32-bit unsigned integer by encoding the year into the most significant 16 bit, the month in the next 8 bit and the day into the least significant 8 bits. [Julian day would be a better encoding.]

Then use a simple hash-function to map a date d to its hash-value by performing

$$d \mod 2^k$$

for some k.

This gives

k	n	#F	#C	llps	#E	avg	uni
8	256	31	31	600	225	589.10	71.34
9	512	62	62	300	450	294.55	35.67
10	1024	124	124	150	900	147.27	17.83
11	2048	247	247	100	1801	73.94	8.92
12	4096	366	366	50	3730	49.90	4.46
13	8192	366	366	50	7826	49.90	2.23
14	16384	366	366	50	16018	49.90	1.11
15	32768	366	366	50	32402	49.90	0.56
16	65536	366	366	50	65170	49.90	0.28
17	131072	731	731	25	130341	24.98	0.14
18	262144	1461	1461	13	260683	12.50	0.07
19	524288	2922	2922	7	521366	6.25	0.03

where $n = 2^k$ is the hash-table size, `#F` is the number of filled entries in the hash-table, `#E` is the number of empty entries in the hash-table, `#C` is number of entries with collisions, `llps` is the length of the longest probe sequence, i.e., the collision chain length, `avg` is the average number of dates falling into one entry, `uni` is the expected number of elements falling into one entry if the hash-functions distributes the dates uniformly.

The murmur hash function can be implemented as follows:

```
template<typename Tuint> Tuint murmurhash(Tuint x);
template<>
uint32_t
murmurhash(uint32_t x) {
    x ^= x >> 16;
    x *= 0x85ebca6b;
    x ^= x >> 13;
    x *= 0xc2b2ae35;
    x ^= x >> 16;
    return x;
}
```

```
template<>
uint64_t
murmurhash(uint64_t x) {
    x ^= (x >> 33);
    x *= 0xFF51AFD7ED558CCD;
    x ^= (x >> 33);
    x *= 0xC4CEB9FE1A95EC63;
    x ^= (x >> 33);
    return x;
}
```

Using murmur hashing, we get:

k	n	#F	#C	llps	#E	avg	uni
8	256	256	256	92	0	71.34	71.34
9	512	512	512	55	0	35.67	35.67
10	1024	1024	1024	34	0	17.83	17.83
11	2048	2048	2047	21	0	8.92	8.92
12	4096	4052	3856	14	44	4.51	4.46
13	8192	7294	5332	10	898	2.5	2.23
14	16384	10973	5044	8	5411	1.66	1.11
15	32768	14018	3545	6	18750	1.3	0.56
16	65536	15905	2147	5	49631	1.15	0.28
17	131072	17015	1194	3	114057	1.07	0.14
18	262144	17637	621	3	244507	1.04	0.07
19	524288	17936	325	3	506352	1.02	0.03

Exercises:

1. Same experiment as the first one, with $d \mod p$ for some prime p. Primes are better.
2. Implement other hash functions, perform the same experiment, measure time for hash function calculation.

4.1.2 Properties

Properties wanted:

1. uniformity
2. universality
3. efficiency

4.1.3 Uniformity

The expected average collision chain length is about n/m where n is the number of keys and m is the hash-table size.

4.1.4 Average-case Search Length

Denote by α the fill-degree $\alpha := n/m$. Then

- on average: successful search: $\Theta(1+\alpha)$
- on average: unsuccessful search: $\Theta(1+\alpha)$

(details see Knuth [153] or Corman [71])

4.1.5 Expected Length of the Longest Probe Sequence (llps)

(see Gonnet [103]) Let n be the number of keys, m the hash-table size, and $\alpha = n/m$ the fill-degree, and $i^k = i(i-1)\dots(i-k+1)$ the descending factorial.

For a full hash-table using uniform probing:

$$E[\text{llps}] \approx 0.631587454 * m + O(1)$$

where m equals the hash-table size and the number of entries.

For a partially filled hash-table using uniform probing:

$$\begin{aligned} E[\text{llps}] &= \sum_{k\geq 0}(1 - \prod_{i=0}^{n-1}(1 - \frac{i^{\underline{k}}}{m^{\underline{k}}})) \\ &\approx -\log_\alpha(m) - \log_\alpha(\log_\alpha(m)) + O(1) \end{aligned}$$

Exercise:

1. Plot $E[\text{llps}]$ for different α, m ignoring the term $O(1)$.

4.1.6 Universality

(see Corman et al [71] and Carter, Wegman [55])

We start with universal.

Let A and B be two sets. A hash-function maps A to B, i.e.,

$$f : A \longrightarrow B$$

A is the set of *potential* keys. We assume $|A| > |B|$.

Let f be a hash-function and $x, y \in A$ two keys. We define

$$\delta_f(x,y) = \begin{cases} 1 & \text{if } x \neq y \text{ and } f(x) = f(y) \\ 0 & \text{else} \end{cases}$$

x and y *collide* under f iff $\delta_f(x,y) = 1$.

In case f, x, and/or y are replaced by a set, this denotes summation. For example

$$\delta_H(x,S) = \sum_{f \in H} \sum_{y \in S} \delta_f(x,y)$$

Def. Let H be a class of functions from A to B. H is *universal* iff $\forall x, y \in A$

$$\delta_H(x,y) \leq |H|/|B|$$

Thus, no *two* distinct keys collide under more than $(1/|B|)$th of the hash-functions.

Proposition 1 shows that the bound on $\delta_H(x,y)$ in the definition of universal is tight.

Prop. 1. For all classes H of hash-functions there exists $x, y \in A$ such that

$$\delta_H(x,y) > |H|(\frac{1}{|B|} - \frac{1}{|A|})$$

Proof. Define $a := |A|$, $b := |B|$. Let $f \in H$.

For each $i \in B$ define $A_i := \{a | a \in A, f(a) = i\}$ and $a_i := |A_i|$.

Note that for $i, j \in B$, $i \neq j$, $\delta_f(A_i, A_j) = 0$. (because elements of A_i are mapped to i and those in A_j to j.)

Note that every element in A_i collides with every other element in A_i. Thus

$$\delta_f(A_i, A_i) = a_i(a_i - 1)$$

Hence,

$$\begin{aligned} \delta_f(A,A) &= \sum_{i \in B} \sum_{j \in B} \delta_f(A_i, A_j) \\ &= \sum_{i \in B} \delta_f(A_i, A_i) \\ &= \sum_{i \in B} a_i(a_i - 1) \end{aligned}$$

The latter is minimized if all A_i are of the same size, i.e., $a_i = a_j = a/b$ for all i, j. This gives us

$$\begin{aligned}\delta_f(A,A) &= \sum_{i\in B} a_i(a_i-1)\\ &\geq \sum_{i\in B} a/b(a/b-1)\\ &= a(a/b-1)\\ &= a^2(1/b-1/a)\end{aligned}$$

Thus, (summing over H)

$$\delta_H(A,A) \geq |H|a^2(1/b-1/a)$$

The left-hand side sums over fewer than a^2 non-zero elements (as $x = y$ implies $\delta_H(x,y) = 0$). The pigeon hole principle implies that there exist $x, y \in A$, $x \neq y$ such that

$$\delta_H(x,y) > |H|(1/b-1/a)$$

□

Proposition 2 tells us about the average collision chain length (averaged over H).

Prop. 2. Let $x \in A$, $S \subseteq A$, H universal class of hash-functions, $f \in H$ chosen randomly. Then, the mean value of $\delta_f(x,S)$ is at most $|S|/|B|$.

Proof. For the mean value we get:

$$\begin{aligned}\delta_f(x,S) &= \frac{1}{|H|}\sum_{f\in H}\delta_f(x,S)\\ &= \frac{1}{|H|}\sum_{y\in S}\delta_H(x,y)\\ &= \frac{1}{|H|}\sum_{y\in S}\frac{|H|}{|B|} \quad \text{[by Def. universal]}\\ &= \frac{|S|}{|B|}\end{aligned}$$

□

The Class H_1 Let $A = \{0,\ldots,a-1\}$ and $B = \{0,\ldots,b-1\}$. Let $p \geq a$ (!) be prime.

Let $g : Z_p \longrightarrow B$ be a function with

$$|\{y \in Z_p | g(y) = i\}| \leq \lceil p/b \rceil$$

(e.g. $g(z) := z \mod b$)

For any $m, n \in Z_p$, $m \neq 0$ define $h_{m,n} : A \longrightarrow Z_p$ via

$$h_{m,n}(x) := (mx+n) \mod p$$

and $f_{m,n} : A \longrightarrow B$ via

$$f_{m,n}(x) := g(h_{m,n}(x))$$

Finally, define the class H_1 of hash-functions from A to B by

$$H_1 := \{f_{m,n} | m, n \in Z_p, m \neq 0\}$$

Lemma Let H_1 be defined as above. Then $\forall x, y \in A$, $x \neq y$

$$\delta_{H_1}(x, y) = \delta_g(Z_p, Z_p)$$

Proof. Since $p \geq a$, p prime, and $m \neq 0$:

$$h_{m,n}(x) = h_{m,n}(y) \;\; \prec\!\succ \;\; x = y$$

and, hence, for $x \neq y$

$$f_{m,n}(x) = f_{m,n}(y) \;\; \prec\!\succ \;\; g(r) = g(s)$$

for $r := h_{m,n}(x)$ and $s := h_{m,n}(y)$. Thus,

$$\delta_{H_1}(x, y) = \delta_g(Z_p, Z_p)$$

□

Theorem 1 H_1 is universal.
Proof.
We have to show that

$$\delta_{H_1}(x, y) \leq |H_1|/|B|$$

Note that $|H_1| = p(p-1)$. Using the lemma, it remains to show that

$$\delta_g(Z_p, Z_p) \leq p(p-1)/b$$

[remember $b = |B|$]
Define $n_i := \{t \in Z_p | g(t) = i\}|$. Then, by definition of g,

$$\forall i \;\; n_i \leq \lceil p/b \rceil$$

Since p and b are integers

$$\lceil p/b \rceil \leq ((p-1)/b) + 1$$

Now, consider some $r \in Z_p$. Then the number of choices for some s with

1. $s \neq r$
2. $g(s) = g(r)$

is limited to $(p-1)/b$.

Since there are p choices for r

$$p(p-1)/b \geq \delta_g(r,s)$$

Recalling $\delta_{H_1}(x,y) = 0$ for $x = y$ and the above concludes the proof. □

Remark: the modulo function is expensive. For Mersenne primes, the modulo operation can be implemented quite efficiently [55].

Let $p = 2^j - 1$ be prime and $x < 2^{2j} - 1$. Let x_1 be the j most significant bits and x_2 the j least significant bit. Then

$$\begin{aligned} x &= 2^j x_1 + x_2 \mod p \\ &= x_1 + x_2 \mod p \end{aligned}$$

since $2^j \equiv 1 \mod p$.

Thus, the following procedure calculates the remainder modulo $p = 2^a - 1$ for some $x < 2^{2a}$.

MOD_MERSENNE(x, p, a)

1 $r = ((x \mathbin{\&} p) + (x \gg a))$
2 **return** $((r < p)\ ?\ r\ :\ (r-p))$

On the XU-4 this is about a factor of three faster than the built-in modulo for 32-bit integers and about a factor of four for 64-bit integers. On the i7-4790 the corresponding factors are 2.5 and 1.5. The exact numbers are compiler dependent.

4.1.7 k-Universal

A class H of hash-functions from A to B is k-universal iff

- for any k distinct elements $a_1, \ldots, a_k \in A$ and
- for any k (not necessarily distinct) elements $b_1, \ldots, b_k \in B$

we have

$$|H|/(|B|^k)$$

functions to map $a_i \to b_i$ for all $i = 1, \ldots, k$.

Or for uniformly random $i \in 1, \ldots, |H|$

$$Pr[h_i(a_1) = b_1, \ldots, h_i(a_k) = b_k] \leq 1/|B|^k$$

4.1.8 (c,k)-Universal

A family $\{h_i\}_{i \in I}$ of hash-functions from A to B is (c,k)-universal iff

- for any k distinct elements $a_1, \ldots, a_k \in A$,
- for any k (not necessarily distinct) elements $b_1, \ldots, b_k \in B$, and
- for uniformly random $i \in I$

we have

$$Pr[h_i(a_1) = b_1, \ldots, h_i(x_k) = y_k] \leq c/|B|^k$$

4.1.9 Dietzfelbinger: Universality without Primes

Dietzfelbinger proposes the following universal class of hash-function [83]. Let $u, k, m \geq 1$ be arbitrary integers with $k \geq u$. Let $U := \{0, \ldots, u-1\}$ and $M := \{0, \ldots, m-1\}$ Define

$$\mathcal{H} := \{h_{a,b} | 0 \leq a, b \leq km\}$$

with

$$\begin{array}{rcl} h_{a,b} & : & U \to M \\ h_{a,b}(x) & := & ((ax+b) \mod km) \div k \end{array}$$

Then, $\mathcal{H}$ is (c,2)-universal with $c = \frac{5}{4}$ [55, 256].

We can look at Dietzfelbinger's hash function as follows. Let the numbers in U and M be at most 32 bits wide. Let a and b be 64-bit integers. Then $(ax+b)$ gives us 33 useful bits $(ax+b)[32..64]$ and we easily extract 32 of them by a simple rightshift. Note that most processors provide instructions directly obtain the higher w word of the result of multiplying two words of width w. See for example Intel's *MULH* instructions.

An efficient implementation of Dietzfelbinger's hash functions using floating point arithmetics is proposed in [247].

4.1.10 k-universal Hash Functions

In order to get k-universal hash functions, we use polynomials.

- Let $A = \{0, \ldots, a-1\}$ and $B = \{0, \ldots, b-1\}$.
- Let $p \geq a$ (!) be prime.
- Let c_i be random numbers in $[0, p[$ for $i = 0, k-1$.

Define H to contain all hash functions h_{c_i} defined as

$$h_{c_i}(x) := \sum_{i=0}^{k-1} c_i x^i \mod p$$

Then H is k-universal.

The same trick with the polynomial also works with the Dietzfelbinger hash function from above [83].

If 2-universal suffices for the application, we can use multiplication-shift-based hashing. Assume we need to hash l_{in} bit numbers to l_{out} bit numbers. Pick some $l \geq l_{\text{in}} + l_{\text{out}} - 1$. Then define H to contain for any $a, b \in [0, 2^l[$ the hash functions defined as

$$h_{a,b}(x) := (ax+b) \gg (l - l_{\text{out}})$$

4.1.11 Tabulation Hashing

Assume we have q hash-functions $h_0, \ldots, h_{q-1} \in H$. Each hash function implemented as an array h_i of random numbers. Assume we hash a value x composed of q (sub-) values x_i (e.g. 4 byte int, string) by

$$\vec{h}(x) := h_0[x_0] \otimes h_1[x_1] \otimes \ldots \otimes h_{q-1}[x_{q-1}]$$

Then, if H is 2-universal then $\vec{h}$ is 2-universal. If H is 3-universal then $\vec{h}$ is 3-universal. After 3, the scheme breaks down.

4-universal hash functions can be build according to the following principle:

$$\vec{h}[x_0x_1] = h_0[x_0] \otimes h_1[x_1] \otimes h_2[x_1 + x_2]$$

For the general scheme: to produce k-universal hash-functions for strings of length q,

$$(k-1)(q-1)+1$$

k-universal hash-functions are required. For more on Tabulation Hashing see [210, 249, 250, 251].

4.1.12 Hashing string values

Ramakrishna, Zobel [217] discuss several hash-functions for strings. Let $s = c_1, \ldots, c_m$ be a string of m characters, v a seed and h_i some intermediate hash value generated after hashing i characters. Then, the generic code of a string hash function is:

```
HASH(s, v)
1  h_0 = INIT(v)
2  for (i = 1; i < m; ++i)
3        h_i = STEP(i, h_{i-1}, c_i)
4  return FINAL(h_m, v)
```

Ramakrishna and Zobel then propose the following class of hash-functions:

$$\begin{aligned} \texttt{init}(v) &= v \\ \texttt{step}(i,h,c) &= h \otimes ((h \ll L) + (h \gg R) + c) \\ \texttt{final}(h,v) &= h \mod T \end{aligned}$$

where T is the hash-table size and L and R are constants with $4 \leq L \leq 7$ and $1 \leq R \leq 3$ where they used $L = 5$ and $R = 2$ in their experiments.

Almost equally good is Larson's string hash function.

Exercise:

1. Compare the above hash-function's performance with Larson's hash-function. As the string values you can use a string-based encoding of dates.

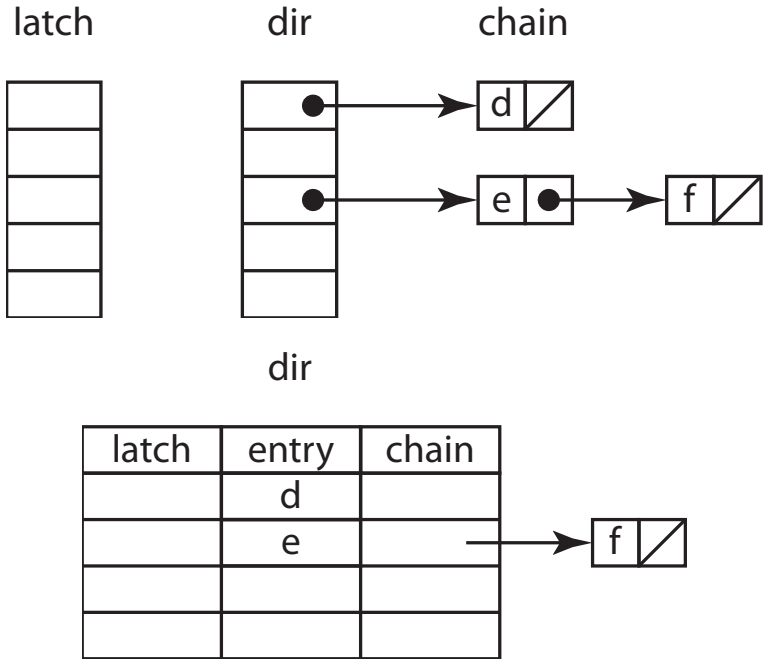
Pearson's [212].

4.2 Hash Table Organization

From A&D:

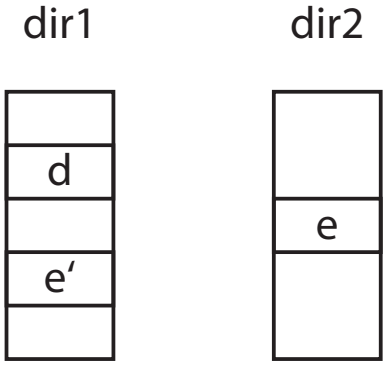
- chaining (may preserve locality for the first element, see below)
- open addressing
  - linear probing
    (preserves locality)
  - quadratic probing
    (does not preserve locality)

4.2.1 Two versions of Chained Hashtable



4.2.2 Cuckoo-Hashing

Like in a cuckoo's nest: the new element kicks out the older element, which in turn is stored in the next level of hash tables:



4.2.3 Robin-Hood-Hashing

4.2.4 Hopscotch-Hashing

Chapter 5

Compression

light-weight compression techniques:

1. zero suppression
2. prefix suppression
3. frame of reference
4. dictionary compression

result: fixed length unsigned integers

see the following papers and the references therein: AODB [257], BLINK [27], HANA [208, 94, 156], using SIMD: [173]

Chapter 6

Storage Layout

6.1 Row stores and Column Stores

Subsequently, we consider the possible storage layouts for the following relation:

eno	name	salary
001	Müller	1000
002	Maier	2000
003	Schmidt	4000

6.1.1 Row format (NSM)

We can concatenate all the bytes for every attribute of a tuple and then concatenate all the tuple's bytes. This results in a *row format*:

001	Müller	1000	002	Maier	2000
003	Schmidt	4000			

This format is also called NSM (N-ary Storage Model).

Row-Format in C++:

```
struct   emp_t {
         int          _eno;
         std::string  _name;
         double       _salary;
};
std::vector<emp_t> Employees;
```

Note: std::string is a performance killer and is not inlined as in the figure.

Exercise:

1. Design a row-format where the string _name is inlined.

6.1.2 Column format (DSM)

Alternatively, the DSM (Decomposed Storage Model) storage layout can be used. Here, every attribute is stored in a binary relation. The first attribute of

this relation contains a surrogate (row identifier (rid) or tuple identifier (tid)) and the second attribute contains the original attribute's value. Here is how DSM looks like for our small relation:

	eno
0	001
1	002
2	003

	name
0	Müller
1	Maier
2	Schmidt

	salary
0	1000
1	2000
2	4000

This storage model was proposed by Copeland and Koshafian [69] and is used in Monet [41]. The original relation can be restored using a join over the rid. The `rid` can be *virtual* and need not be stored explicitly. Then, we arrive at the *column format*:

eno
001
002
003

name
Müller
Maier
Schmidt

salary
1000
2000
4000

Again, Monet is one of the forerunners applying this storage format [41].

Column-Format in C++:

```
struct   Employees {
         std::vector<int>          _eno;
         std::vector<std::string>  _name;
         std::vector<double>       _salary;
};
```

Exercise:

1. discuss different alternatives for std::string for string columns (e.g., string containers without dictionary, with unordered dictionary, with ordered dictionary)

Query processing example

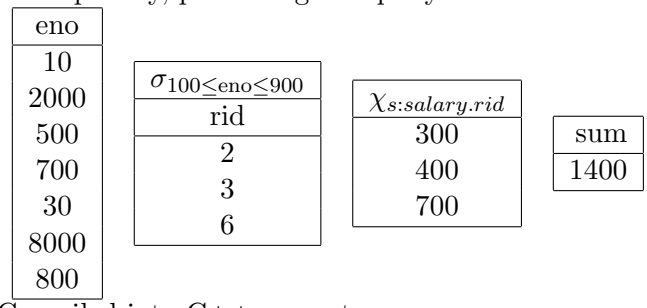
Query processing: example: Bigger table `emp`:

rid	eno	name	salary
0	10	—	100
1	2000	—	200
2	500	—	300
3	700	—	400
4	30	—	500
5	8000	—	600
6	800	—	700

stored columnwise (rid implicit as index into column array).

select **sum**(salary)
from Employees
where eno between 100 and 900

Conceptually, processing the query looks as follows:



Compiled into C++, we get

```
int sum = 0;
for(size_t i = 0; i < emp.eno.size(); ++i) {
    if((100 ≤ emp.eno[i]) && (emp.eno[i] ≤ 900))
        sum += emp.salary[i];
}
return sum;
```

Insert example

insert into Employees **values** (333, "Trump", 33)

```
Employees::insert(int e, std::string n, double s) {
      _eno.push_back(eno)
      _name.push_back(n)
      _salary.push_back(s)
}
```

Exercise:

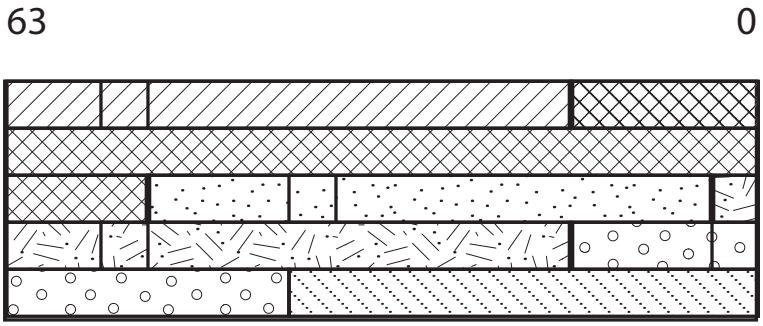
1. implement deleting a tuple with a given rid.
2. implement an index on Employee.eno

6.1.3 Hybrid Storage Model (PDSM)

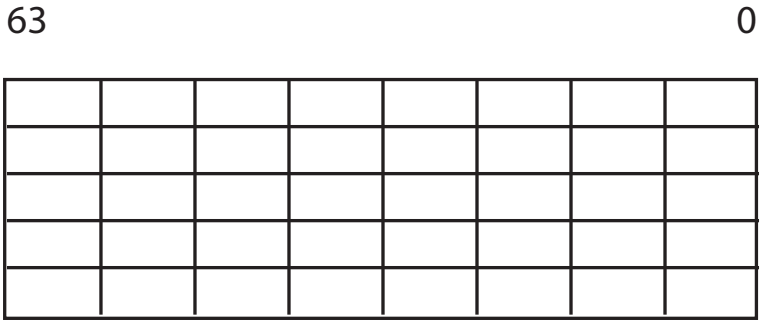
It is obvious, that we can decompose a relation not only into binary relations but arbitrarily. This results in the *partially decomposed storage model* [123, 213]. Attributes used frequently together are then stored together in one fragment.

6.1.4 Cache Lines in Row and Column Format

Obviously, the storage layout impacts cache utilization.
Things look bad for the row store:



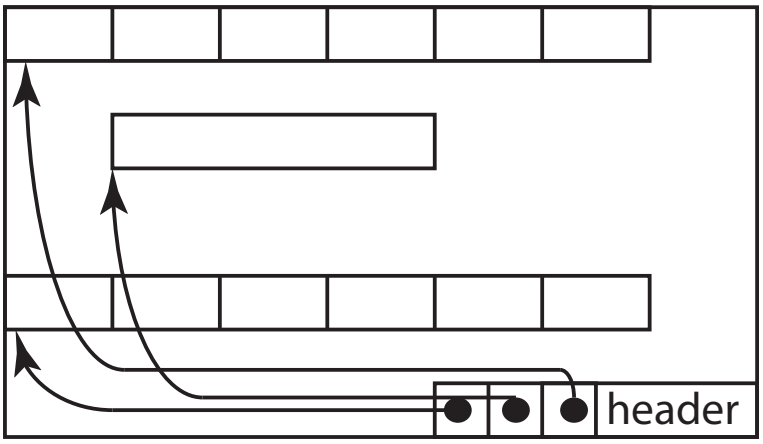
Things look good for the column store:



6.2 Organization on Pages

Both row and column store formats can organize their data in very large chunks of bytes. This is typically not practical if updates occur, indices are applied etc. In DBSI, we thus learned about an organization of the row storage format on pages called *slotted pages*. Together with the *tuple identifier* concept, this allowed for highly flexible updating/moving around of tuples.

Analogously, several small columns can be organized onto a page. One proposal to do so is PAX [7]. A PAX page looks as follows:



This looks very similar to a slotted page. The only difference is that instead of pointing to tuples, the slots contain pointers to arrays of attribute values, i.e., a column. There are some further details not discussed here.

Exercise:

1. How can we deal with variable-length fields in PAX.
2. How can we deal with nullable attributes.

6.3 Row Layouts

fixed length: easy. complications:

1. variable length fields
2. null-values
3. compression

To keep attribute values aligned, we assume that records are aligned to, say, 8 bytes. Then, we put all the 8-byte attributes at the beginning (e.g., doubles d_j), followed by the 4-byte attributes (e.g., integers i_j), followed by 2-byte, and finally 1-byte attributes:

d_1	d_2	i_1	i_2	i_3

Adding variable size attribute values, for example strings s_i, is rather simple: We add in the fixed-length part offsets to the strings. Note: o_1 points to the start of s_1 and is the end of s_0. A last o_{k+1} denotes the end of s_k. (end = one character after the last). Adding three string values results in:

d_1	d_2	i_1	i_2	i_3	o_0	o_1	o_2	o_3	s_0	s_1	s_2

Dealing with NULL-values, we have two possibilites:

- reserve some special value to represent NULL-values
- add NULL-indicators

The former approach it applicable only in special cases, e.g., for dictionary compression where a dictionary id of 0 is reserved for NULL-values. In general, the latter case must be supported.

Adding NULL-indicators (`nid`):

nid	d_1	d_2	i_1	i_2	i_3	o_0	o_1	o_2	o_3	s_0	s_1	s_2

Disadvantage: still space allocated for NULL-attributes. We now eliminate the wasted space. A consequence is that offsets to attributes are no longer the same for every tuple as different tuples may have NULL-values in different attributes. Assume in one tuple d_1 is NULL and i_1 is NULL. The layout then is:

$1010\ldots0$	d_2	i_2	i_3	o_0	o_1	o_2	o_3	s_0	s_1	s_2

If we assume d_1, d_2, i_3, and s_1 to be NULL, we get

$11001010\ldots0$	i_1	i_2	o_0	o_2	o_3	o_4	s_0	s_2

There are several possibilities to calculate the offset of an attribute, some with layout changes, some not:

1. interprete the null-indicator: go through the bits of the null-indicate and perform offset calculation. This means access to an attribute becomes linear in the time of nullable attributes. If we always store the not-null attributes followed by the nullable attributes (for each attribute size), we can restrict this overhead to nullable attributes.

2. use an offset array within each record similar to the variable size attributes for null-able attributes. if offsets are smaller than actual values this saves some space.

3. use a separate table where these offset calculations are materialized

4. use `uval_t` arrays as tuples. Using `popcnt` on the null-indicators up to the attribute to be accessed and subtract this from the attribute number to be accessed. Of course, `uval_t` arrays waste some memory.

5. The same `popcnt` solution can be used if null-indicators are grouped by attribute size.

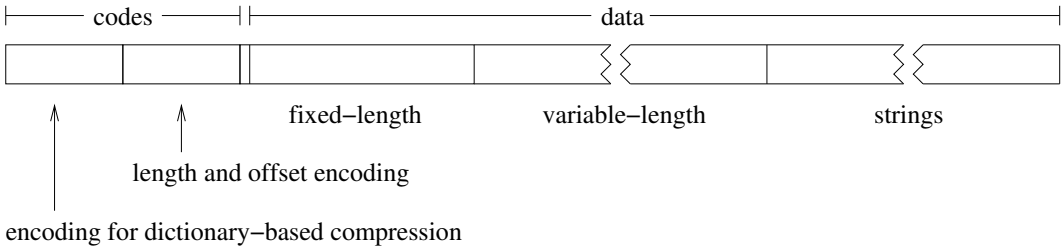
Remark:

- Whereas `uval_t` arrays may be used during query execution, they are never used as a storage layout of tuples.

- The table based approach (discussed in [257]) has the disadvantage that every table consumes a few kilobyte of memory, which puts additional stress onto the L1d cache. Further, access latency to L1d is about 3-4 cycles whereas `popcnt` has a latency of 3 cycles.

Compression adds another layer of complexity. Assume we add leading-zero-suppression for integers. If we restrict the length of integers to multiples of a byte, integers can now be 0, 1, 2, 3, or 4 bytes long. (0 bytes in case of '0'). The size of a compressed integer may vary in every tuple. In this case, the only studied approach is an offset-table-based one [257].

The basic record layout there is:



For every attribute with variable length (including compressed and nullable attributes), we use *status bits* to encode its length and, possibly, null-status. For example:

4 byte integers		
length	NOT NULL	nullable
0	–	000
1	00	001
2	01	010
3	10	011
4	11	100

8 byte floats		
0	-	00
4	0	01
8	1	10

These status bits are packed together within bytes such that always all status bits belonging to a certain attributes are contained in one byte. Unused hi-bits are set to zero. Consider for example a relation with attributes

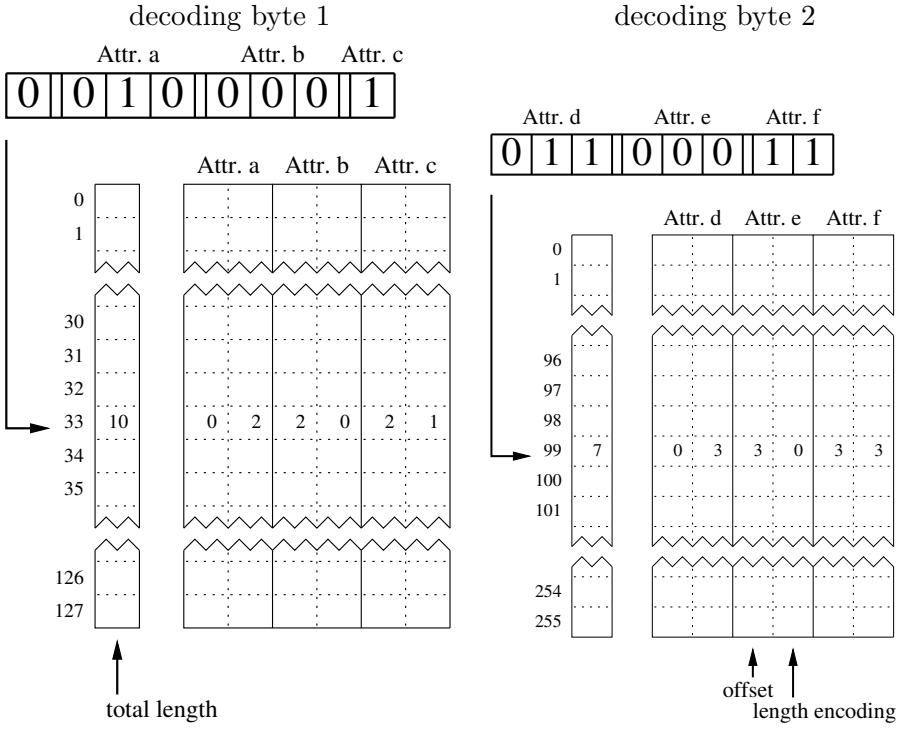
(a int, b int, c double not null, d int, e int, f int not null)

Assume all attributes are compressed. Then, all attributes become variable length attributes and two bytes are necessary for the length encodings:

–	b_1^a	b_2^a	b_3^a	b_1^b	b_2^b	b_3^b	b_1^c
b_1^d	b_2^d	b_3^d	b_1^e	b_2^e	b_3^e	b_1^f	b_2^f

The following table shows an example. For every attribute, there are two things stored:

1. the offset
2. the length encoding (and not the length itself)



The code to calculate the offset of some variable-length attribute is

```
int off(int     attrno,      // number of attribute
        int*    codeBytes,   // code bytes of row
        int     byteNo,      // number of code byte for attr
        dct     table) {     // decoding table
  int off = 0;
  for(int j = 0; j < byteNo; ++j)
     off += table[j][codeBytes[j]].total;
  return off + table[byteNo][codeBytes[byteNo]].offset(attrno)
       + table[byteNo][codeBytes[byteNo]].length(attrno);
}
```

Remark: we omitted one important aspect in the above discussion, which is versioning. In a real system, tuples must be versioned. The reason is that schema changes may take place. For example, we might add or remove attributes. If we are not able to version our tuples, a schema change requires to modify all tuples immediately. With versioning, tuple layout change implied by schema changes can be performed lazily.

6.4 Column Layouts

- simple: array of fixed length values (int32, int16)
- often: compression, e.g., by
  - ordered dictionary or
  - dates via julian day and frame of reference compression.
- consequence: fewer than 32 or 16 bits needed per value
- wish: use only as few bits per value as possible
- more compression, e.g., runlength encoding (possibly after sorting, etc.)

Note: the same complications arise as in the row-store case.

6.4.1 BitPackingH

This is the original storage format used by Hana [259, 260].

Using dictionary compression or some other technique, assume that n bits suffice to store any attribute value of some attribute A. Then, values of A are represented as a bitvector of length $n * |R|$. Example ($n = 3$):

a	a	a	b	b	b	c	c	c	d	d	d	e	e	e	f	f	...

Decoding:

- unpack into 32-bit integers
- implementation using SIMD instructions

Decoding Steps (128 bit SIMD):

1. 16 Byte alignment: make sure 128-bit registers start with complete compressed value. Assume currently handled value is in the upper part of a 256-bit register
   (a) load second 128-bit register into lower part of a 256-bit register
   (b) perform a 256-bit register shift
2. 4 Byte alignment:
   (a) apply a shuffle operation to put four consecutive compressed values into the 4 32-bit words of a 128-bit register
3. Bit alignment:
   (a) apply a shift operation with 4 individual shifts
   (b) apply a bitwise AND operation with a mask to zero out irrelevant bits

For details see [260, 259].

Problems:

- comparisons for selection predicate (e.g. `between`) after decompression
- improvement: it is possible to insert the selection predicate evaluation after the first few steps of the decompression algorithm (comparison can be done before bit alignment, by shifting the constants with which to compare accordingly.)

6.4.2 BitSliceH

BitSliceH and BitSliceV (discussed in the next section) were both proposed by Li and Patel [177]. The latter builds on bitslicing originally proposed by Rinfret, O'Neil, O'Neil [224].

BitSliceH uses one bit more than necessary. It is set to zero. Consider again the case of $n = 3$ bits necessary to encode a value. Then the BitSliceH storage layout looks like

0	a	a	a	0	b	b	b	0	c	c	c	0	d	d	d	0	...

This one additional bit is used to hold the result of comparison operations. Further, no codes span multiple lines (a line corresponds in length to a SIMD register and is accordingly aligned). Instead, padding is used.

We discuss how comparison of a column with a value can be implemented (see [177] for details). Let w be the SIMD register length. Let X be the SIMD register holding $w/(k+1)$ column values. Let Y be the SIMD-register holding $w/(k+1)$ times the value with which the column is to be compared. Let Z be the result vector where the additional bit indicates the result of the comparisons. Let x and y be two k bit values.

Further we will use the following bitwise operators:

- $\barwedge$ bitwise and

- $\oslash$ bitwise or
- $\otimes$ bitwise xor
- $\neg$ bitwise complement

Inequality We have $x \neq y$ iff $x \otimes y \neq 0^k$. Adding 01^k to 01^k does not produce an overflow. Thus, Z can be calculated as

$$Z = ((X \otimes Y) + 01^k 01^k \dots 01^k) \oslash 10^k 10^k \dots 10^k$$

Equality complement of inequality

$$Z = \neg((X \otimes Y) + 01^k 01^k \dots 01^k) \oslash 10^k 10^k \dots 10^k$$

Less Than

$$\begin{array}{lrcl} & x & < & y \\ \iff & x & \leq & y - 1 \\ \iff & 2^k + x & \leq & y + 2^k - 1 \\ \iff & 2^k & \leq & y + 2^k - 1 - x \end{array}$$

Note that $2^k - 1 - x = \neg x = x \otimes 1^k$. Thus (no overflow can occur):

$$Z = (Y + (X \otimes 01^k 01^k \dots 01^k)) \oslash 10^k 10^k \dots 10^k$$

Less Than Or Equal To Since $x \leq y$ iff $x < y + 1$ we have

$$Z = (Y + 0^k 1 \dots 0^k 1 + (X \otimes 01^k 01^k \dots 01^k)) \oslash 10^k 10^k \dots 10^k$$

Careful: y must be less than $2^k - 1$. This does not do any harm since $2^k - 1$ is the maximum possible value and thus a comparison $x \leq 2^k - 1$ would always yield true.

Greater $>$ and $\geq$ can be implemented using $<$ and $\leq$ and an argument swap.

Indicator Bit Extraction Methods to extract the indicator bits are described in Appendix B of [178]. The first method is rather simple: successively shift/or/mask. Let $b = k + 1$ be the length of one block of bits. Every such block is the form $c0^k$ where the bit c indicates the comparison result. After one of the comparison operators defined above, the result is of the form

$$c_1 0^k \dots c_m 0^k$$

which we wish to transform into

$$c_1, \dots, c_m, 0^*.$$

where $m = 2/(k+1)$.

$$\begin{array}{lrcl} \text{Step 1:} & Y & = & (X \veebar (X \ll 1(b-1))) \barwedge (0^{2b-2}1^2 \ldots 0^{2b-2}1^2) \\ \text{Step 2:} & Y & = & (Y \veebar (Y \ll 2(b-1))) \barwedge (0^{4b-4}1^4 \ldots 0^{4b-4}1^4) \\ \text{Step 3:} & Y & = & (Y \veebar (Y \ll 4(b-1))) \barwedge (0^{8b-8}1^8 \ldots 0^{8b-8}1^8) \\ \ldots & & & \end{array}$$

Example:

$$\begin{array}{l|cccccccc|cccccccc|} \text{Input:} & c_1 & 0 & 0 & 0 & c_2 & 0 & 0 & 0 & c_3 & 0 & 0 & 0 & c_4 & 0 & 0 & 0 \\ \text{Step 1:} & c_1 & c_2 & 0 & 0 & 0 & 0 & 0 & 0 & c_3 & c_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{Step 2:} & c_1 & c_2 & c_3 & c_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Li and Patel also propose a second alternative method [178] based on multiplication, which is only applicable if $b \leq \sqrt{w}$:

$$Y = (X * (0^{b-2}10^{b-2}1 \ldots 0^{b-2}1)) \barwedge (1^{\lfloor w/b \rfloor}0^{\lfloor w/b \rfloor (b-1)})$$

The method is based on the idea that shifting is the same as multiplication. All shifts are performed at once here. However, one must be sure that these don't interfere. That is the reason for the condition $b \leq \sqrt{w}$.

Since the multiplication method is faster than the shift-based method, it should be used if applicable and the shift-based method should be used in the remaining cases.

Remark: On POWER processors one can use specialized instructions for bit-permutations.

Converting Bit Vectors to Indices (RIDs) A method to extract the indices of the indicator bits is described in Appendix A of [178]: Define two helper functions:

$$\begin{array}{lcll} \text{rlsb}(x) & := & x \barwedge (x-1) & \text{// reset least-significant bit set} \\ \text{smsb}(x) & := & x \otimes (-x) & \text{// set to most-significant bits up to the lsb set} \end{array}$$

The intrinsic `blsr` implements rlsb with one machine instruction;

Example:

		0	1	2	3	4	5	6	7	msb
x	=	0	1	1	0	1	1	0	1	
rlsb(x)	=	0	0	1	0	1	1	0	1	
smsb(x)	=	0	0	1	1	1	1	1	1	

Algorithm:

- loop over all bits set in a word x in the bitvector
- for all bits set: determine their index and output it after adding some base.

Assumption: the index of the most significant bit is the lowest index.

```
INPUT: BV: input bitvector, w: word width
OUPUT: L: vector of RIDs
p = 0
foreach x in BV
    while(x ≠ 0)
        rid = p + popcnt(smsb(x)) // get base + index
        L += rid // append rid to output L
        x = rlsb(x) // reset least significant bit set
    p += w // add word length to base p
return L
```

Alternative: use bit-scan-forward/reverse to extract index of a lowest/highest bit set.

Remarks (1) In case there are no spare bits between encoded values in the storage format, we could do the following to implement comparisons on the compressed data: Comparison on compressed data:

- split word into even numbered and odd numbered parts (2 masks)
- do stuff as with 0-bits inbetween
- join together both results

This requires that the most significant bit is left unused.
(2) in order to evaluate $A \geq c$ for some attribute A and constant c, we could also set the indicator bits to '1' in the encdoding of A, subtract c, and mask on the indicator bits.

6.4.3 BitSliceV

6.4.4 ByteSliceV

ByteSliceV is the latest proposal to organize tuples in storage [96].

6.5 DB2 BLU

In this section, we briefly describe storage layout of DB2 with BLU Acceleration (DB2 BLU for short) [90], which builds on (predicate evalution in) BLINK [144, 218]. Note however, that DB2 BLU (as seen below) is (mainly) a column store whereas BLINK is row store.

6.5.1 Column Groups

Let R be a relation. For every attribute $A \in \mathcal{A}(R)$ which may contain NULL-values, a *null-indicator attribute* is added. The attributes $\mathcal{A}(R)$ of a relation R can be partitioned into *column groups*. Any attribute A which may contain NULL-values and its null-indicator attribute must be contained in the same column group.

- Column groups are stored on pages.
- Pages are allocated in chunks called *extents*.
- Each extent contains data from one column group only.
- Tuple Sequence Numbers (TSN) are used to identify tuples.
- For every tuple, the TSN is the same in each column group.
- A tuple projected on the attributes of a colum group is called *tuplet*.
- Each page contains a page header.
- A page header contains a StartTSN and a TupleCount.
- A *page map* is used to map a (columngroup,TSN) pair to a page. It is implemented as a B^+-Tree.

6.5.2 Compression

Standard compression technique, but the active domain of an attribute can be partitioned into several subdomains. This partitioning is best be done frequency based. That is, highly frequent values go to one partition and values of low frequency to another. The compression scheme may then differ for each partition. For example, a different number of bits maybe used to encode the values in different partitions.

Example:

- We compress 16 bit country codes in a trading database.
- We partition the country codes into three partitions.
  - We use 1 bit compression for China and Russia.
  - We use 3 bits for other countries with a lot of trading.
  - We use 8 bits for the remaining countries

In general, the distinct values of an attribute A of some relation R are partitioned into a few partition. For every such partition, a separate order-preserving dictionary is allocated [218]. Details on how to determine the partitions are also contained in [218].

6.5.3 Cell/Region

The space of possible formats of the tuplets in a column group is determined by the cross product of

- the partitions of all columns of a column group

These combinations are called *cells*. Within a page, all tuplets belonging to the same cell and (thus) have the same format are stored together in a *region*.

6.5.4 Page Format

A page contains the following elements:

1. page header
2. page-specific compression dictionaries
3. regions stored in banks
4. tuple map
5. variable width data bank

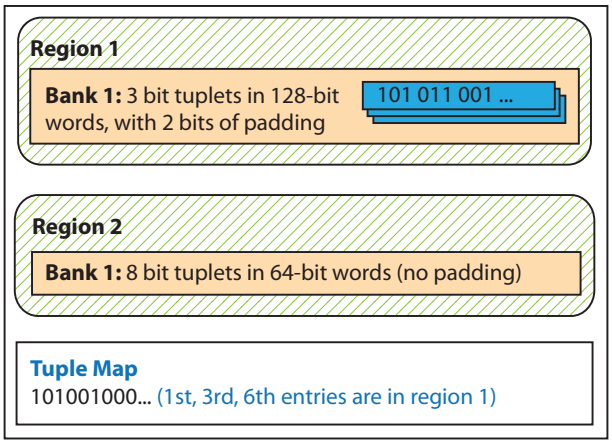
We discuss those items not already discussed subsequently.

If a page contains more than one region, it contains a *tuple map* which records to which region a tuple belongs. The *tuple map* is indexed by the page-relative TSN and contains as many bits as necessary to uniquely determine a region.

Regions are subdivided into *banks*. Banks are contiguous areas of the page that contain the actual tuplet values (or their encoding). For example, assume there is an attribute A which may contain NULL-values. Then, the actual values of A and the A's null-indicator column may be stored in separate banks.

Most banks contain fixed size tuplets. For compressed values, the encoded tuples are stored packed together in 128-bit or 256-bit words with padding to avoid straddling of tuplets across word boundaries. The 128 or 256 is called *width* of the bank. Word lengths 8, 16, 32, 64 bits can also be used.

The following picture contains a page. This page stores columns of a single-column column group: country-code. Although there exist altogether three regions for country code (as explained above), only two regions are present on the given page.



Considering only banks of one cell per attribute group and abstracting from pages and regions, we have the following image for a relation with attributes a, $\ldots$, g vertically partitioned into column groups $\{a, b, c\}$, $\{d, e, f\}$, {g,h}.

2	9	10	11
-	a	b	c
-	a	b	c
-	a	b	c
-	a	b	c
-	a	b	c

Bank 1
w=32

1	8	10	13
-	d	e	f
-	d	e	f
-	d	e	f
-	d	e	f
-	d	e	f

Bank 2
w=32

2	9	5
-	g	h
-	g	h
-	g	h
-	g	h
-	g	h

Bank 3
w=16

In this figure, the i-th tuple consists of the i-th word of every bank. The '-' sign indicates unused bits. Further, the tuplet size coincides with the width of the bank. In general, this does not need to be the case: several tuplets may be contained in a w bit wide bank.

Fixed-width uncoded values are also stored in banks of regions where the bank width is determined by the data type of the attribute.

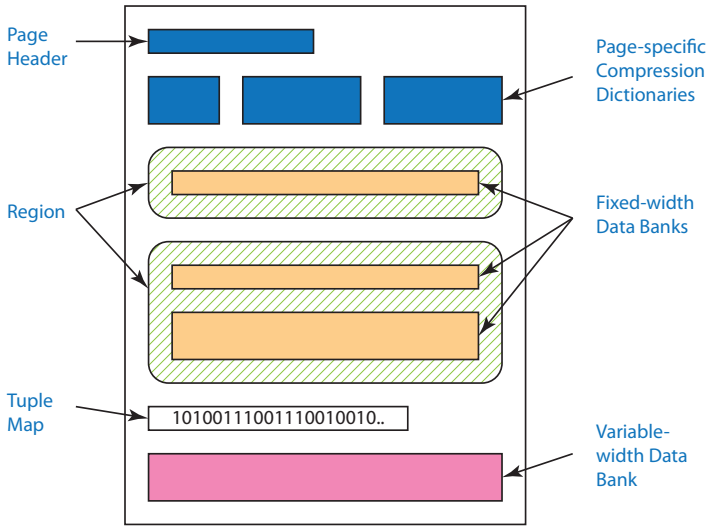
Uncoded variable-width values are stored consecutively in a separate variable-width data bank. Additionally, a pair (offset,length) is stored in a regular (fixed-width) bank.

6.5.5 Page Compression

Page level compression exploits the fact that only a subset of the codes maybe present in tuples to be stored on a given page. In this case, a smaller number of bits might suffice. Thus, if beneficial, mini-dictionaries are stored on a page. These then contain the page-level compression scheme.

Adjusting the frame of refererence in frame-of-reference compression can also be done at the page level.

All elements of a page are contained in the following figure:



6.5.6 Small Materialized Aggregates (SMA)

Small internal synopsis tables are created automatically for every colum-organized table. These contain the following small materialized aggregates:

- MinTSN, MaxTSN for every page
- min/max values of columns contained in the page

A synopsis table uses the same compression scheme as the regular column-organized table for which it were created.

6.5.7 Global Code

The compression dictionary contains the

- partition-relative encoding and
- a global coding.

Assume the partitions are ordered into $P_1, \ldots, P_n$. Then, the i-th local code of P_k gives rise to the global code

$$(\sum_{l=1,\ldots,k-1} |P_l|) + i$$

The global code is used in join and grouping operations.

6.5.8 Scan

Evaluating a selection predicate in multiple attributes on the DB2 BLU storage format is not an easy task. It is performed in mainly three steps:

1. SCAN-PREP: scan synopsis, apply predicates to synopsis to skip pages
2. LEAF: scan one horizontal partition and apply predicates, collect TSNs of qualifying tuples.
3. LCOL: for the other columns not contained in the column group of LEAF access these columns using the TSNs.

This evaluation requires some infrastructure like TSNlists to represent qualifying TSNs in a compact form. Predicate evaluation is performed for multiple simple comparison predicates for different columns in a tuplet i parallel (see [144] for details). The bit-operations are very similar to those in BitSliceH [177]. A complication arises if multiple cells occur in a page. In this case, the results from these pages must be interwoven using the tuple map (see details in Sec.5.1.2 of [90]).

6.6 SQL Server

6.6.1 Apollo

The SQL Server column store started out as additional user-definable column indices. Later, column only attributes became possible [163, 88]. We briefly discuss the storage layout of the SQL Server column store.

As the column store started out as an additional index, copies of the columns could extracted on already populated tables and stored separately in a columnar format. This allowed for a *row group* wise processing. For every row group, the attribute values of a certain attribute where extracted and a dictionary populated. For each of the row groups and the dictionary, the data is then encoded and compressed. Each compressed column in then stored in a *segment*. The dictionary also contains other metadata like the number of tuples in the segment, size of the segment, kind of the encoding, min/max values.

Segments are stored in BLOBs. The segments are not stored in the regular buffer pool but instead in a special large-object buffer pool. This allows to store the pages of a segment as continous pages in buffer with no 'page breaks'.

SQL Server uses two dictionaries per column:

1. global dictionary for the whole column
2. local dictioinaries for each row group

The global dictionary is optional.

The tow group to column creation process works in three steps:

1. encoding
2. determine optimal row order
3. compress each column

SQL Server uses two types of encodings:

1. dictionary-based encoding (maps values to 32 or 64 bit integers)
2. value-based encoding (convert integers/decimals to integers)

Since SQL Server supports run-length encoding sorting achieves the best compression result. Thus, by looking at the possible compression ratios for the different columns some overall best row order can be determined.

For compressing a column there is a choice between run-length encoding and bit packing.

Updates must be handled using a delta.

6.6.2 Hekaton

Hekaton is the SQL Server main memory row store. Several papers describe various aspects of it [12, 82, 86, 97, 162, 138, 174].

The goal of Hekaton was to improve the throughput of SQL Server by 10x-100x. The following back of the envelop calculation showed this is not possible by small code rewrites [82].

The performance of any OLTP system can be expressed as

$$\mathrm{SP} = \mathrm{BP} * \mathrm{SF}^{\log_2(N)}$$

where

SP	=	system performance
BP	=	performance of a single core
SF	=	scalability factor
N	=	number of cores

Using

IR	=	instructions retired
CPI	=	cycles per instruction

we can rewrite the above to

$$\mathrm{SP} = \mathrm{IR} * \mathrm{CPI} * \mathrm{SF}^{\log_2(N)}$$

Observations on SQL Server on some common OLTP benchmarks:

- CPI of less than 1.6 (which is fairly good)
- SF is 1.89 up to 256 cores (which is also fairly good)

At 256 cores SQL Server throughput increases by factor of

$$1.89^8 = 162.8$$

Ideally, a factor of 256 would be achieved. Thus, the maximal possible improvement factor gainable from increasing SF is

$$256/162.8 = 1.57$$

Further, an extraordinarily good CPI 0.8 would give another factor of 2 improvement for SP. Giving a total of

$$2 * 1.57 = 3.14$$

Thus, to achive 10x-100x a drastic decrease (90% to 99%) of IR is necessary!

Analysis of existing systems shows where the time executing a request is spent [126, 143]. Based this experience, the Hekaton design team came up with the following architectural principles for Hekaton:

1. optimize indexes for main memory
   (classical B-tree lookup: thousands of instructions)
2. eliminate latches and locks
   (latch-free data structures, optimistic multi-version concurrency control)
3. compile into native code

Storage and indexing of Hekaton tables:

- Hekaton table is completely contained in main memory
- two types of indexes:
  - Bw-Tree (latch-free [138])

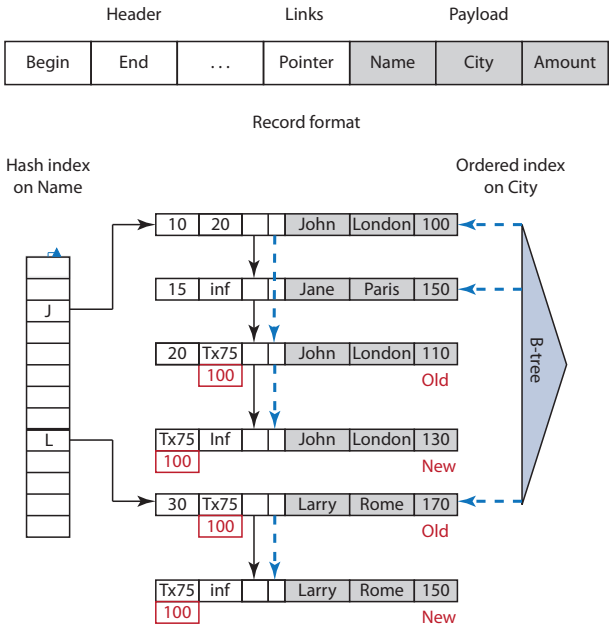


Figure 6.1: Storage Layout in Hekaton

  - hash index (latch-free [190])

- a table can have multiple indexes
- record lookup is always by indexes

Fig. 6.1 shows the storage layout of a hekaton table containing bank account data. The table contains 6 versioned records. The record format is shown at the top of the figure.

- Name, City, Amount: regular attributes of the relation
- begin/end: validity interval
- link fields: one per index chaining entries for that index

In the example there exist two indices. The hash index on name takes for implicitly as a hash function only the first character of the name. Collisions are chained via the black pointers in the figure. The second index is a B-Tree, giving rise to the blue pointers which link together records with the same key.

Read. Reading is performed for a specific time. For any time only one version of a record qualifies.

Update. Assume transaction 75 transfers 20 Yen from Larry's account to John's account. This creates a new version of both account records. The old version receive a 75 as their end-timestamp and the new versions have a

75 in their begin-timestamps. At commit time, both these begin- and end-timestamps are updated to the timestamp at which transaction 75 commits. Assume this is 100. Then, the 75 entries in begin- and end-timestamps are replaced by 100. For details on transaction management see [82].

6.7 Large Objects

starburst long field manager, exodus long field manager etc.

Chapter 7

Physical Algebra: Processing Modes

7.1 Pull Algebra

Traditional pull-based algebra interface (as in DBSI):

- open
- next
- close

`next` is called once per tuple.

7.2 Push Algebra

7.2.1 Interface

Producer interface:

- run

Consumer interface:

- init
- step
- fin

`step` is called once per tuple.

7.2.2 Scan

Sample code for scan:

```
class Scan : public Producer {
    void run(Segment S) {
        foreach page P in S {
            foreach tuple T on page P {
                _consumer->step(T)
            }
        }
    }
    Consumer* _consumer;
};
```

7.2.3 Select

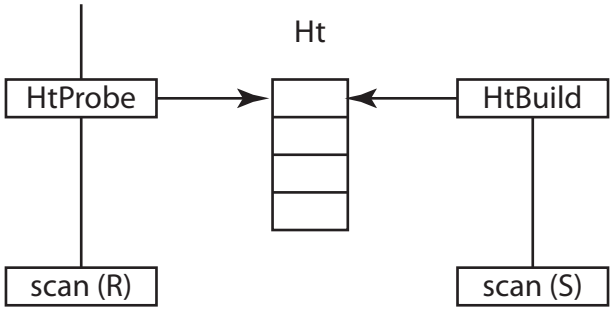
Sample code for selection:

```
class Select : public Consumer {
    void step(Tuple T) {
        if((*_predicate)(T))
            _consumer->step(T);
    }
    Consumer* _consumer;
    Predicate* _predicate;
};
```

7.2.4 Simplest Hash-Join

The hash-join is split into two parts:

1. build (build hash table)
2. probe (probe other relation and build result tuples)



Evaluation of $R \bowtie^{hj} S$ proceeds in two steps:

1. execute `run` on build relation (S)
2. execute `run` on probe relation (R)

Pseudocode: For simplicity, we assume that

- the argument to the step function is a rid (i.e., the type of Tuple is uint) and every function knows how to access the right parts of the tuple.

- the hash functions h_r and h_s are somehow known and return an unsigned int (uint)

- the hash functions h_r and h_s take a rid as argument and implicitly know where to find the join attributes.

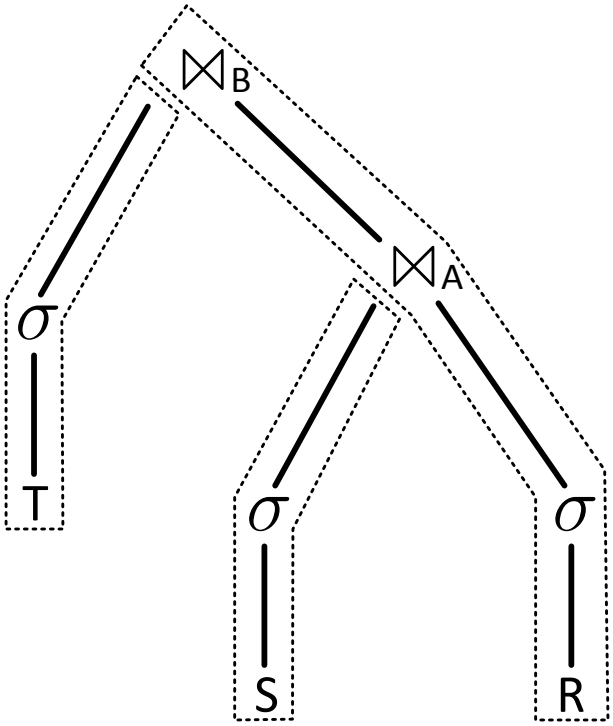
- we only store the rid of the tuple in the hash table

- all required functions work with rids

- the result of the join is represented as pairs of rids of the joining tuples represented by two aligned vectors Sres, Rres.

```
typedef std::unordered_map<uint, std::vector<uint>> hashtable_t;
class HJoinBuild {
    void step(Tuple s) {
        _ht[h_s(s)].push_back(s);
    }
    hashtable_t _ht;
}
class HJoinProbe {
    void step(Tuple r) {
        for(auto s : _ht[h_r(r)]) {
            if(JoinPredicate(r, s)) {
                Rres.push_back(r);
                Sres.push_back(s);
            }
        }
    }
    hashtable_t& _ht;
}
```

In general, some order must be observed when executing strands. In the following figure, there are three strands. Here, the build input is on the left-hand side of every join:



Discussion:

- push-based algebraic operators are easier to implement than their pull-based counterparts
- needs some runtime coordination: strands
- push-based algebra are good for code-generation (one code-fragment per strand)
- push-based algebra has low overhead (when compiled)

Exercise:

1. discuss the simplifying assumptions
2. discuss the case of a key-foreign-key-join
3. discuss how template classes for hash-table functions and instances thereof containing pointers to columns might help to implement a generic version with little code.
4. what happens if the join predicate is an equi-join over several attributes? how can this case be handled?
5. how can we implement non-equi joins, outerjoins, semijoins, antijoins, groupjoins?
6. how can we implement the different set operations and duplicate elimination?

7.3 Materialization Granularity/Call Granularity

7.3.1 Tuple-wise (single tuple materialization)

As in the above code, per call to next/step one tuple is processed. This results in some performance penalties:

- function call overheads: next/step and predicate/subscript
- lack of code locality (L1i misses)

The advantage is that there is only one tuple to be materialized. That is, the memory can be reused for every tuple processed (except for pipeline breakers (see DBSI)).

7.3.2 Complete (full materialization)

An alternative is that every operator of the physical algebra produces a completely materialized result. This disadvantage here is that a huge amount of memory is needed and likewise a fair amount of memory-bandwidth.

7.3.3 Blockwise (partial materialization)

This is the middle way. In each call to next/step a bunch of tuples is processed. Memory for this bunch has to be allocated (best: if it fits into some cache). The size of a bunch of tuples can be determined by the number of tuples contained in it or by some size in bytes.

Two alternatives are possible for pipelining blocks/chunks/bunches:

- one input bunch of tuples produces one output bunch of tuples.
  This simplifies the logic but, e.g. after a selection, the ouput bunch of tuples may contain very few tuples (overhead is back).
- many input bunches of tuples can produce one output bunch of tuples.

Remark: In case of full and blockwise materialization, we can materialize either in row or in column format, independently of the input storage format.

Exercise:

1. sketch the code for push-algebra with full materialization.
2. sketch the code for push-algebra with blockwise materialization.

Chapter 8

Expression Evaluation

8.1 Introduction

Several operators take subscripts/functions/programs which must be evaluated. For example: selection predicates, join predicates, projection lists, map-expressions.

some operators may take several subscripts/programs: e.g., the hash-join operator:

- calculate hash-function for right input
- calculate hash-function for left input
- calculate result of join predicate
- concatenate two input tuples

In a push-based algebra, it is rather simple to compose complex expressions which evaluate a sequence of pipelined algebraic operators (*strand*):

- scan-[select,map,semijoin,antijoin,project]-mat

Such a complex program would be given to the `scan` operator.

In general there are two possibilities to evaluate these expressions: interpretation and compilation. For each of them, we have different sub-possibilities:

- interpretation
  - operator tree with `eval`
  - virtual machine
- compilation
  - C or similar
  - LLVM [197, 198, 48]
  - machine code [160, 180]

Before we discuss these possibilities, let us look at alternative result representations.

8.2 Result Representation

- tuples in any of the storage layouts (col,row,...)
- and additionally
  - to represent the result of a selection:
    - list of indices (pointers/rids/tids) of qualifying tuples
    - bitvector of qualifying tuples
  - to represent result of join:
    - pairs of indices (pointers/rids/tids)

8.3 Interpretation: Operator Tree

Every operation for every supported type is encapsulated within a class. The common superclass has the interface

```
typedef unsigned char byte_t;
class SimpleOpBase {
     virtual byte_t* eval() = 0;
     SimpleOpBase* _args[MAXARGS];
}
```

To avoid `byte_t` pointers, one can define a union-type `uval_t` containing the `union` of all supported types (and more):

```
typedef    union {
           int32_t    _i32;
           double     _f64;
           ...
} uval_t;
```

Exercise:

- change the interface of SimpleOpBase using `uval_t`
- implement a few operators with both interfaces
- discuss the pros and cons of each approach

8.4 Interpretation: A Virtual Machine (AVM)

All virtual machines need some instruction set:

```
enum   avm_instr_e {
       kAvmStop      =   0,
       kAvmAddI32    =   1,
       kAvmSubI32    =   2,
       kAvmMulI32    =   3,
       kAvmDivI32    =   4,
       kAvmModI32    =   5,
       kAvmEqI32     =   6,
       ...
       kAvmNoOp      =   MAXNOOP
};
```

A *program* is a sequence of `uint32_t` reflecting a sequence of op-codes followed by arguments:

1. op-code from `avm_instr_e`
2. zero or more arguments in the form of attribute numbers or offsets into row-tuples depending on the storage layout.

Putting together a program (here for row format):

```
uint32_t     lProg[7];
lProg[0]  =  kAvmEqI32;
lProg[1]  =  0; // offset of arg 1
lProg[2]  =  4; // offset of arg 2
lProg[3]  =  kAvmStop;
```

8.4.1 AVM: row: single tuple

Two general approaches: switch vs. function pointers.

This program is then interpreted by some interpreter functions, e.g.

```
/*
*     a simple avm interpreter processing one tuple in row format at a time
*     aTuple: pointer to tuple
*     aProg: pointer to avm program
*     this one is implemented with a switch statement
*/
int   avm_itp_row_single_switch(byte_t* __restrict__ aTuple, uint32_t* __restrict__ aProg);
/*
*     a simple avm interpreter processing one tuple in row format at a time
*     aTuple: pointer to tuple
*     aProg: pointer to avm program
*     this one is implemented with an array of function pointers
*/
int   avm_itp_row_single_funptr(byte_t* __restrict__ aTuple, uint32_t* __restrict__ aProg);
```

Here is how the variant using a switch-statement looks like:

```
#define OP(a1, a2, a3, op, T) (*(T*)(a1)) = (*(T*)(a2)) op (*(T*)(a3))
int
avm_itp_row_single_switch(byte_t* __restrict__ t, uint32_t* __restrict__ p) {
  int lRes = 0;
  byte_t *a1, *a2, *a3; // pointers to attribute values
  LOOP:
    switch(*p++) {
      case kAvmStop :  goto END;
      case kAvmAddI32:
        a1 = t + *p++; // add offset to tuple base pointer
        a2 = t + *p++;
        a3 = t + *p++;
        OP(a1, a2, a3, +, int32_t);
        break;
      ...
      case kAvmEqI32:
        a1 = t + *p++; // add offset to tuple base pointer
        a2 = t + *p++;
        lRes = ((* (int*) a1) == (* (int*) a2));
        break;
      ...
    }
  goto LOOP;
  END:
  return lRes;
}
```

For the variant using function pointers, we first need an array of function pointers:

```
    typedef int (*op_fun_t)(byte_t* a, byte_t* b, byte_t* c);
op_fun_t gOpFunArr[] = { 0, &fun_addi32, &fun_subi32, &fun_muli32,
                  &fun_divi32, &fun_modi32 };
```

where the functions `fun_XXX` have to be implemented somewhere. Just to give an example:

```
      int  fun_addi32  (byte_t* a, byte_t* b, byte_t* c) {
                       OP(a, b, c, +, int32_t);
                       return 0;
      }
```

Using the array of function pointers, we can code an interpreter:

```
int
avm_itp_row_single_funptr(byte_t* __restrict__ aTuple, uint32_t* __restrict__ p) {
    int lRes = 0;
    byte_t* __restrict__ t = aTuple;
```

```
    int lOp = 0;
    LOOP:
      lOp = *p++;
      if(kAvmStop == lOp) {
        goto END;
      }
      lRes = (gOpFunArr[lOp])((t + *p), (t + *(p+1)), (t + *(p+2)));
      p += 3;
      goto LOOP;
    END:
    return lRes;
}
```

Exercise:

1. Implement a push-based selection using a pointer to a program to evaluate its selection predicate.
2. How can we deal with operations of different arity in the function pointer case?
3. How can we deal with NULL-values?

8.4.2 AVM: row: vectorized

The general problem with the above approach is that for every tuple there is a call to the interpreter and for every operation within the program there is either a goto (switch) or a function call. This overhead is compared to the time it costs to, e.g., add two numbers, gigantic.

The idea is to reduce this overhead by moving the loop over tuples into the interpreter. Thus, a single add-operation now copes with many tuples instead of one. This helps amortizing the goto/function-call overhead.

Again, we have two possibilities: switch and function pointers.

t tuple pointer

n number of tuples

w tuple width

p program

```
int
avm_itp_row_vectorized(byte_t* t, int n, int w, uint32_t* p) {
    int i;
    byte_t* *a1, *a2, *a3; // pointers to attribute values
    LOOP:
      switch(*p++) {
        case kAvmStop :  goto END;
        case kAvmAddI32:
```

```
            a1 = t + *p++; // get pointers to attributes
            a2 = t + *p++; // by adding offsets
            a3 = t + *p++; // contained in avm program
            for(i = 0; i < n; ++i) {
              OP(a1, a2, a3, +, int32_t);
              a1 += w; // advance attribute pointers by tuple width
              a2 += w; // advance attribute pointers by tuple width
              a3 += w; // advance attribute pointers by tuple width
            }
            break;
          ...
        }
    goto LOOP;
    END:
    return n;
}
```

Exercise:

1. Implement a switch-based vectorized expression evaluator.
2. Discuss the problem occuring with selection predicates in the vectorized context. How could it be solved?
3. Discuss the problem occuring with varying tuple width in the vectorized context. How could it be solved?

8.4.3 AVM: col: single

Here, instead of offsets into tuples, the arguments of every AVM-operation are column indices identifying a column. Here is a switch-based implementation:

```
int
avm_itp_col_single(byte_t* aColPtrs[], int aTupleNo, int* p) {
    int lRes = 0;
    byte_t *a1, *a2, *a3; // pointers to attribute values
    LOOP:
    switch(*p++) {
      case kAvmStop :  goto END;
      case kAvmAddI32:
        a1 = aColPtrs[*p++] + (aTupleNo * sizeof(int32_t));
        a2 = aColPtrs[*p++] + (aTupleNo * sizeof(int32_t));
        a3 = aColPtrs[*p++] + (aTupleNo * sizeof(int32_t));
        OP(a1, a2, a3, +, int32_t);
        break;
      ...
    }
    goto LOOP;
    END:
```

```
    return lRes;
}
```

8.4.4 AVM: col: vectorized

```
int
avm_itp_col_vectorized(byte_t* aColPtrs[],
                       const int aStartRid,
                       const int aNoTuples,
                       int* p) {
   byte_t *a1, *a2, *a3; // pointers to attribute values
   LOOP:
   switch(*p++) {
       case kAvmStop :  goto END;
       case kAvmAddI32:
           a1 = aColPtrs[*p++] + (aStartRid * sizeof(int));
           a2 = aColPtrs[*p++] + (aStartRid * sizeof(int));
           a3 = aColPtrs[*p++] + (aStartRid * sizeof(int));
           for(int i = 0; i < aNoTuples; ++i) {
              OP(a1, a2, a3, +, int32_t);
              a1 += sizeof(int);
              a2 += sizeof(int);
              a3 += sizeof(int);
           }
           break;
   }
   goto LOOP;
   END:
   return n;
}
```

Exercise:

1. Implement a function pointer based version of avm.col.vectorized.
2. Look at the generated assembler code of these functions. Do they contain SIMD-instructions? [can use godbolt]

8.4.5 AVM: col: vectorized with SIMD

Two possibilites:

1. rely on compiler
2. use intrinsics

Normally solution (1) sufficies since the loops are very stylized and the compiler is able to generate SIMD-code. Since the compiler does not know about alignments that maybe guaranteed by the QEE, the code generated is typically a bit more complex and a little less efficient.

8.5 Compilation

C/C++:

- simplest to implement
- results in fast expression evaluation
- compiler call is mostly unacceptably costly

LLVM:

- a little more difficult to implement
- results in fast expression evaluation
- compiler call maybe too expensive, especially for short-running ad-hoc queries

MachineCode/Assembler:

- tedious to implement
- lower 'compilation' overhead
- results in fast expression evaluation
- not portable

8.6 Comparison: simple map program

Evaluation time for a simple program adding/subtracting five integer attribute values and assign the result to some other attribute. More specifically, the program measured corresponds to

$$\text{A[0]} = \text{A[0]} + \text{A[1]} - \text{A[2]} + \text{A[3]} - \text{A[4]}$$

where $A[i]$ denotes the i-th integer attribute. The relation contained a total of 90 integer attributes and no other ones. The following figures contain the runtime results for both row and column stores. Abbreviations used:

rs row store, avm interpreter, single tuple, switch

rs2 row store, avm interpreter, single tuple, function pointer

rsc row store, compiled, single tuple

rv row store, avm interpreter, vectorized

rv2 row store, avm interpreter, vectorized (different implementation)

rc row store, compiled, vectorized

cs column store, avm interpreter, single tuple, switch

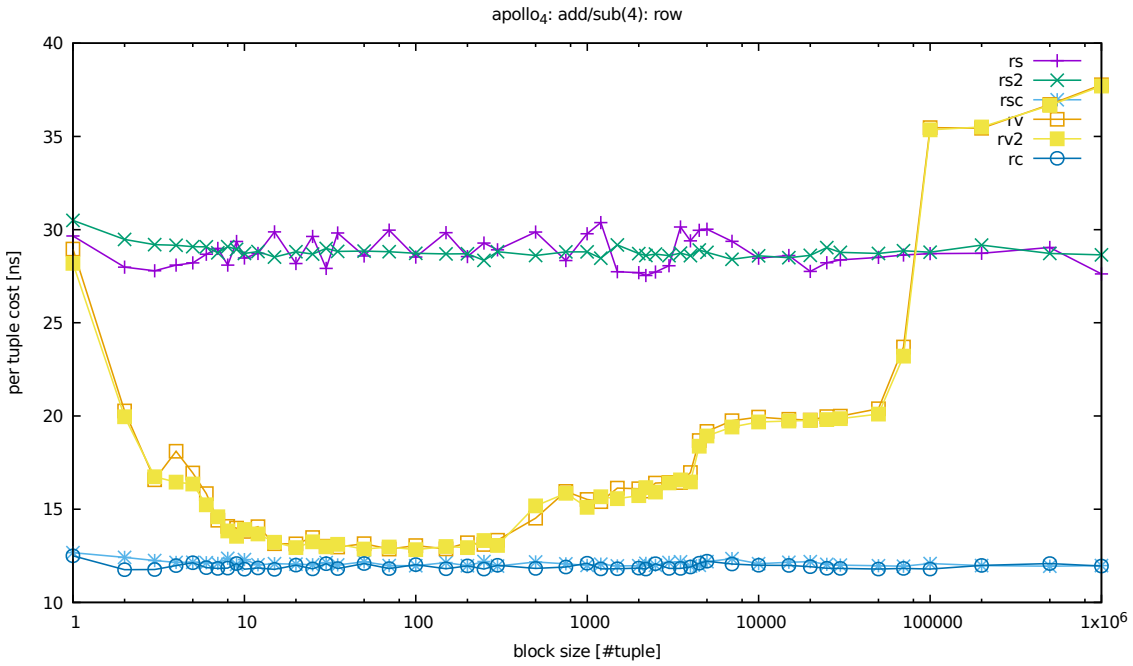
cs2 column store, avm interpreter, single tuple, function pointer

cv column store, avm interpeter, vectorized, switch

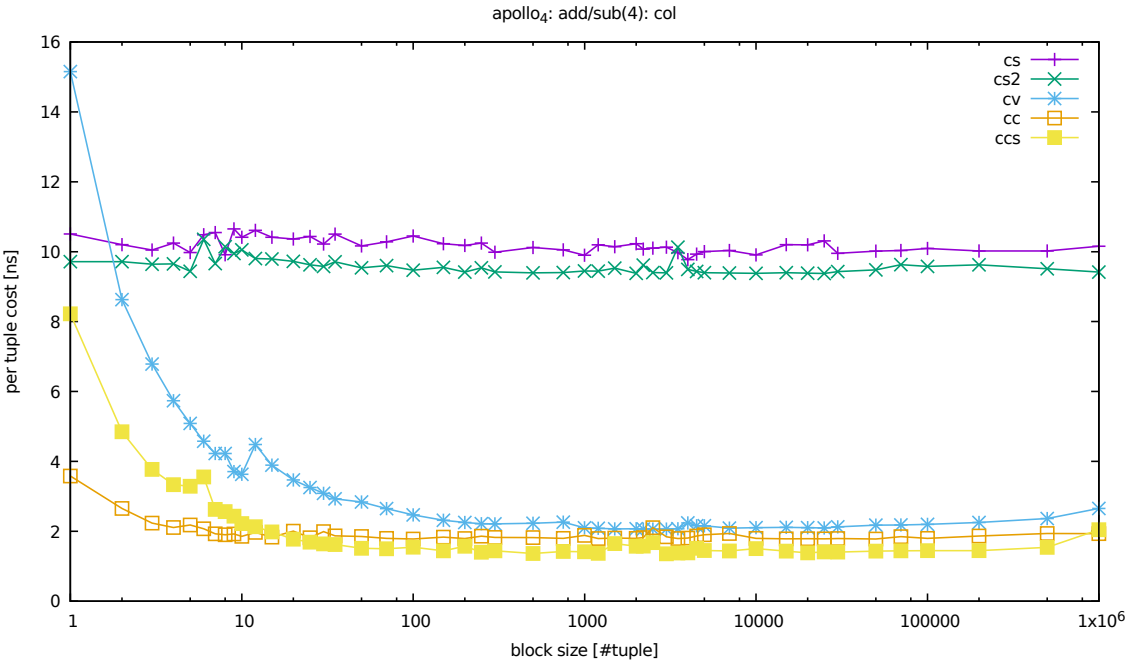
cc column store, compiled, vectorized, no SIMD

ccs column store, compiled, vectorized, with SIMD

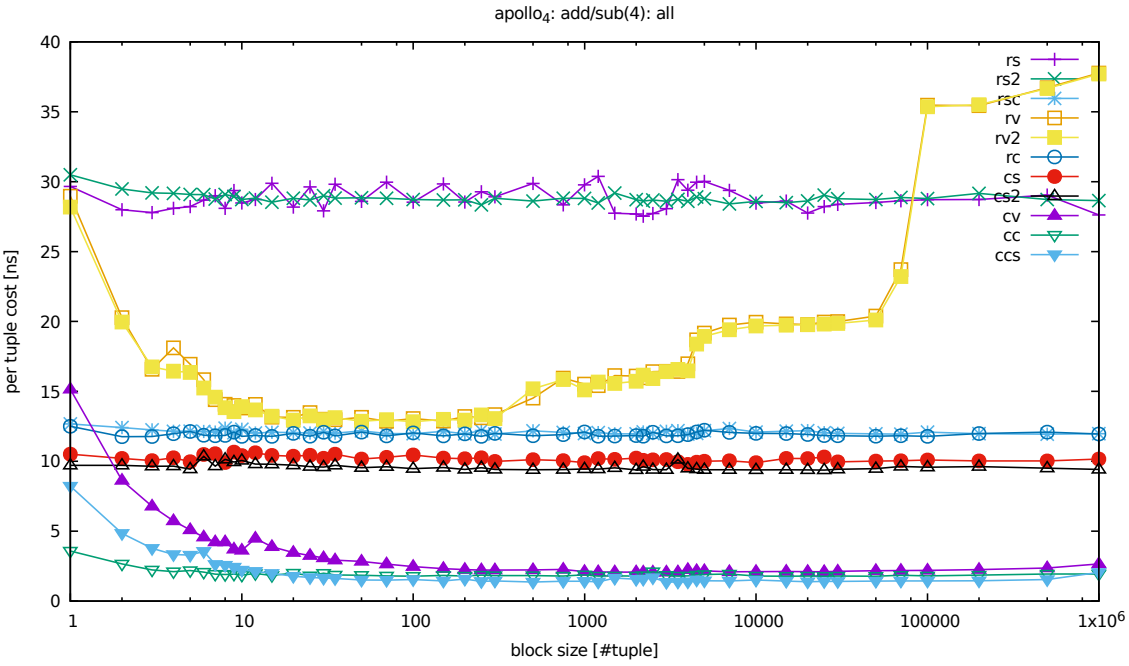
row:



col:



all:



8.7 AVM: col: Vectorized SIMD: selection

During the above discussion it became clear that there is a problem with selection operators under vectorization as not every input tuple produces an output tuple. The output of a selection can be:

- produce column projection, i.e., vectors containing the key column and one or more payload columns
- a vector of indices of qualifying tuples
- a bitvector with '1' for qualifying tuples

We discuss the first possibility and leave the others as an exercise. In the latter two cases, there must be (variants of the) algebraic operators that take an index vector/bitvector as an additional input.

We give Algorithm 3 of [214] to implement a SIMD-based selection with a `between` predicate. In order to avoid using special SIMD-instruction, which makes the code somewhat more difficult to read, we use the notation developed in [214].

- W is the number of entries in one SIMD-register.
  For example: $W = 4$ in case of 4-byte integers in a 128 bit SSE or NEON register.
- To denote a SIMD-register, vector notation is used: $\vec{r}$.
- $\leftarrow$ denotes assignment.
- masked or selective assignment is denoted by $\vec{r} \leftarrow_m \vec{p}$ for a mask m indicating which entries of $\vec{p}$ are copied to $\vec{r}$.

The code uses a *software-managed buffer* B. The idea here is that it remains in the cache and streaming write is used to flush it to main memory.

The following algorithm produces for some input column T_{in} for every index i such that $k_{lb} \leq T_{\text{in}}[i] \leq k_{ub}$ an output column T_{out} containing the qualifying key from T_{in} and an output column P_{out} containing values from a corresponding input column P_{in}. Adding more of these is easy. If only a set of indices is needed as output, the code can be simplified accordingly.

```
SELECT_BETWEEN
i, j, l ← 0                                  // index for in/out/buffer
r⃗ ← [0, ..., W − 1]                          // input indices
for(i = 0, i < |T_in|; i+ = W)               // for each lane
    k⃗ ← T_in[i]                              // read W input values
    m ← (k⃗_lb ≤ k⃗)&(k⃗ ≤ k⃗_ub)                // 'between' to mask
    if(0 ≠ m)                                // at least one qualifying input key?
        B[l] ←_m r⃗                           // selectively store indices
        l ← l + |m|                          // inc each component by |m|
        if(|B| − W < l)                      // buffer almost full?
            for(b = 0; b < |B| − W; b+ = W)  // step through buffer
                x⃗ ← B[b]                     // load idx of qualifying tuples
                k⃗ ← T_in[x]                  // load qualifying keys
                p⃗ ← P_in[x]                  // load qualifying payload
                T_out[j + b] ← k⃗             // store key values
```

```
        P_out[j + b] ← p⃗          // store payload
      p⃗ ← B[|B| − W]              // move overflow ..
      B[0] ← p⃗                    // ..  to buffer begin
      j ← j + |B| − W             // update output index
      l ← l − |B| + W             // update buffer index
  r⃗ ← r⃗ + W                      // update index vector
// after loop:  flush remaining items in buffer
```

Exercise:

1. Implement the code for a SIMD vectorized column-based between-predicate where the result contains only the indices of qualifying input tuples. There is no need for a software-managed buffer here. Measure performance. Hint: Appendix D of [214] contains two SIMD-implementations of the above pseudocode.

2. Implement the code for a SIMD vectorized column-based between-predicate where the result is represented as a bitvector. Measure performance. Hint: See [271, 214, 203]

Chapter 9

Physical Algebra: Implementation

9.1 General Implementation Techniques

When implementing algorithms for a DBMS, the following points have to be taken into account:

- avoid branch-misprediction (e.g. by predicated code)
- avoid interpretation overhead (e.g. by vectorization or compilation)
- avoid cache misses (make algorithms cache conscious)
- avoid TLB misses

The following techniques exist to make algorithms cache conscious [237]:

1. Blocking/Tiling
2. Partitioning
3. Extraction
4. Loop Fusion

A general treatment of cache conscious pointer-based data structures can be found in [64].

Just to have a complete list of techniques, we mention the others here. They are treated in other sections.

1. software managed buffers (see, e.g. Sec. 8.7, Sec. 9.4)
2. explicit prefetching
3. streaming stores (possibly with software write-combining)

9.1.1 Blocking/Tiling

Assume we implement something like a nested loop join where each element from one input is compared to each element with some other input. Further assume that both inputs are available as arrays X and Y. Consider the following code fragment [237]:

for($i = 0$; $i < m$; $++i$)
    for($j = 0$; $j < n$; $++j$)
        process($X[i]$, $Y[j]$)

Assume that Y does not fit into the cache. Then, many cache misses occur in this code. (Alghough in this case it might not be that bad since we have sequential access to Y and the prefetcher helps hiding latencies.) If we process Y in blocks/tiles each fitting into the cache, we can get better cache-locality and, hence, performance. Let B be the block-size chosen such that B elements of Y fit into some cache. Consider the following code where b denotes the block number:

for($b = 0$; $b < n/B$; $++b$)
    for($j = 0$; $j < n$; $++j$)
        for($j = b * B$; $j < (b + 1) * B$; $++j$)
            process($X[i]$, $Y[j]$)

Exercise

1. Discuss the similarities/differences with the regular nested-loop join and the blockwise nested loop join.

9.1.2 Partitioning

Consider a simple `sort` operation of an array X of size n:

`quicksort`(X, n)

Due to the workings of quicksort, this results in many cache-misses if X is large.

An alternative is to *partition* X into small partitions, sort them individually and then merge the results:

partition X into partitions x of size $m <$ cache size
for each partition x
    `quicksort`(x,m)
merge all partitions

This is the essential idea of the in-memory Alpha-Sort [201], an extremely fast sort algorithm.

9.1.3 Extraction

Extracting only the data needed for the next step is often another good idea. Instead of sorting (and thus copying around) whole tuples, it might be beneficial to extract only the (sort-) key and a pointer to a tuple. This miminizes the amount of data to be accessed during the actual sort. If the (sort-) key is long, it might even be beneficial to extract just a prefix (as done in Alpha-Sort [201]). For building a hash-table might be equally beneficial to extract the key (or its hash) and a pointer/rid/tid.

9.1.4 Loop Fusion

If we have two subsequent loops that range over the same data it might be beneficial to apply *loop fusion*. Here, we see extraction and hash-table insert implemented with two separate loops:

for($i = 0$; $i < n$; $++i$)
  $A[i]$.key = relation$[i]$.key
  $A[i]$.ptr = relation$[i]$.ptr;
for($i = 0$; $i < n$; $++i$)
  insert_into_hashtable($A[i]$)

This can be improved by loop fusion as in

for($i = 0$; $i < n$; $++i$)
  $A[i]$.key = relation$[i]$.key
  $A[i]$.ptr = relation$[i]$.ptr;
  insert_into_hashtable($A[i]$)

because most probably $A[i]$ will be in the cache, if not in some register.

Exercise:

1. measure building a hash table containing 100.000 tuples containing each 90 integer attributes
2. apply extraction and measure again

9.2 Scan/Select

We have discussed most alternatives already:

- branching code (Sec. 2.2.3) versus predicated code (Sec. 2.2.3)
- SIMD Sec. 8.7

9.3 Join

We first discuss two improvements (extraction, partitioning) of the simplest hash join from Sec. 7.2.4 as discussed by Shatdal, Kant, and Naughton [237].

9.3.1 Simple Hash Table

The simplest implementation of the hash join builds a complete hash table for the inner R and then probe with the outer S (as in Sec. 7.2.4):

```
HtBuild(H_R, R)
for each s ∈ S
    Probe(s, H_R)
```

Here, whole tuples of R are stored in the hash-table. If R is small (smaller than some cache and TLB is no issue), this algorithm should perform well.

9.3.2 Extraction

A first improvement results from extracting key-pointer-pairs from R and inserting these into the hash-table:

```
for each r ∈ R
    H_R.insert(ExtractKeyPointer(r))
for each s ∈ S
    Probe(s, H_R)
```

This increases locality and if the size of H_R is not too large (cache/TLB), this algorithm should perform well.

9.3.3 Partitioning

The next step is to partition both relations similar to the Grace-Hash-Join discussed in DBSI. We also use extraction for this algorithm:

```
PartitionedHashJoin(R, S)
    Partition(ExtractKeyPointer(R))
    Partition(ExtractKeyPointer(S))
    for each partition i
        HtBuild(H_{R_i}, R_i)
        for each s ∈ S_i
            Probe(s, H_{R_i})
```

If the partitions are rather (very) small, we can use a nested loop join to join the partitions:

```
PartitionedNestedLoopJoin(R, S)
    Partition(ExtractKeyPointer(R))
    Partition(ExtractKeyPointer(S))
    for each partition i
        NestedLoopJoin(R_i, S_i)
```

This avoids the overhead of building a hash table.

This pseudocode leaves out the details of partitioning. These are discussed in the next section.

9.3.4 Software Prefetching

Last in this section let us consider *software prefetching* as a means to accellate the hash-join. Here, the code contains explicit prefetch instructions to make sure that the hash directory entries and collision chain elements are in the cache when needed. We discuss several techniques proposed in the literature.

9.3.5 Group Prefetchin

Chen, Ailamaki, Gibbons, Mowry compare *group prefetching* and *software-pipelined prefetching* as two prefetching strategies. In *group prefetching*, input tuples are partitioned into small groups and the statements in the build/probe-phase are applied to each tuple in a group with prefetching commands included. The following pseudocode illustrates group prefetching for the probe-phase:

```
foreach group of tuples in probe partition
    foreach tuple in the group
        compute hash bucket number
        prefetch the target hash bucket
    foreach tuple in the group
        visit hash bucket header
        prefetch collision chain next (if necessary)
    foreach tuple in the group
        visit the collision chain (if necessary)
    foreach tuple in the group
        visit matching build tuples
            to compare keys and produce output tuple
```

[here: entries consist of hash-value and pointer to tuple]
Disadvantages of group-prefetching:

1. bursts of prefetches
2. complexity

9.3.6 Software-Pipelined Prefetching

Software-pipelined prefetching goes as follows:

```
prologue;
for j=0; j< N - 3D; ++j
    tuple j+3D:  compute hash bucket number
                 prefetch the target bucket header
    tuple j+2D:  visit the hash bucket header
    tuple j+D;   visit the collision chain
                 prefetch the matching build tuple
    tuple j:     visit the matching build tuple
                 compare keys and produce output tuple
epilogue;
```

Disadvantages of software-pipelined prefetching:

1. pipelining in probe too short, even shorter in build
2. complexity

9.3.7 Rolling Prefetching

Hence, we introduce a third alternative, which we call *rolling prefetching*. It has a parameter k which indicate how many intermediate hash codes we store. An implementation for $k = 2$ looks as follows:

```
template<class Tuint, class Tbun, class Thashfun>
void
build_rp_2(const std::vector<Tbun>& aBun) {
    const size_t m = size();
    const size_t n = aBun.size();
    Tuint lIdxA = 0;
    Tuint lIdxB = 0;
    if(2 < n) {
        lIdxA = Thashfun()(aBun[0].key()) % m;
        lIdxB = Thashfun()(aBun[1].key()) % m;
        __builtin_prefetch(&(_dir[lIdxA]), 1, 0); // optional
        __builtin_prefetch(&(_dir[lIdxB]), 1, 0); // optional
        const size_t nx = n - 2;
        for(size_t i = 0; i < nx; ++i) {
            insert_at(aBun[i], lIdxA);
            lIdxA = lIdxB;
            lIdxB = Thashfun()(aBun[i+2].key()) % m;
            __builtin_prefetch(&(_dir[lIdxB]), 1, 0);
        }
        for(size_t i = nx; i < n; ++i) {
            insert(aBun[i]);
        }
    } else {
        build(aBun); // regular build for small relations
    }
}
```

Using local variables lIdxC, lIdxD and so forth, leads to $k = 3$, $k = 4$ and so forth. However, this is a little too tedious (and expensive). Here is an alternative implementation for $k = 8$:

```
template<class Tuint, class Tbun, class Thashfun>
void
build_rp_8(const std::vector<Tbun>& aBun) {
    const size_t m = size();
    const size_t n = aBun.size();
    Tuint lIdx[8];
```

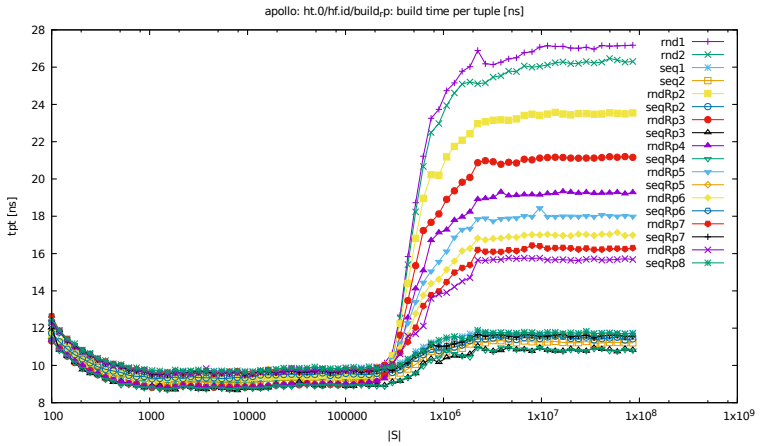


Figure 9.1: Runtime of build with and without rolling prefetch for i7-4790

```
    if(8 < n) {
        for(int i = 0; i < 8; ++i) {
            lIdx[i] = Thashfun()(aBun[i].key()) % m;
            __builtin_prefetch(&(_dir[lIdx[i]]), 1, 0);
        }
        const size_t nx = n - 8;
        const uint32_t lMask = 0x7;
        uint32_t lCurr = 0;
        for(size_t i = 0; i < nx; ++i, ++lCurr) {
            insert_at(aBun[i], lIdx[lCurr & lMask]);
            lIdx[lCurr & lMask] = Thashfun()(aBun[i+8].key()) % m;
            __builtin_prefetch(&(_dir[lIdx[lCurr & lMask]]), 1, 0);
        }
        for(size_t i = nx; i < n; ++i) {
            insert(aBun[i]);
        }
    } else {
        build(aBun); // regular build for small relations
    }
}
```

Figures 9.1, 9.2, 9.3 contain the runtimes of the rolling prefetched versions for different platforms. [hash function: id; seq: sequential key insert, rnd: random key insert; md1/2: regular non-prefetched implementations, Rpk: rolling prefetch k]

Rolling prefetching has the disadvantage that it only prefetches the directory entries. This is sufficient if there exist only very few collisions since one entry is stored within the directory itself. However, if there are collisions, e.g. in case of a non-key join attribute and skew, walking the collision chain will result in expensive L3 misses and main-memory accesses. Hence, we discuss an extension

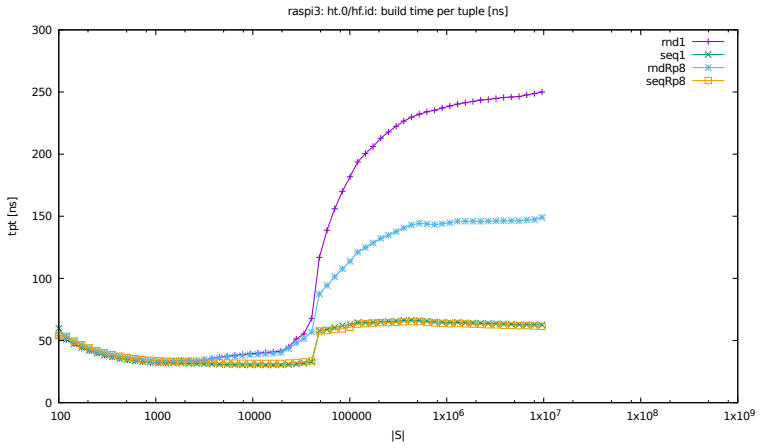


Figure 9.2: Runtime of build with and without rolling prefetch for Raspberry Pi 3

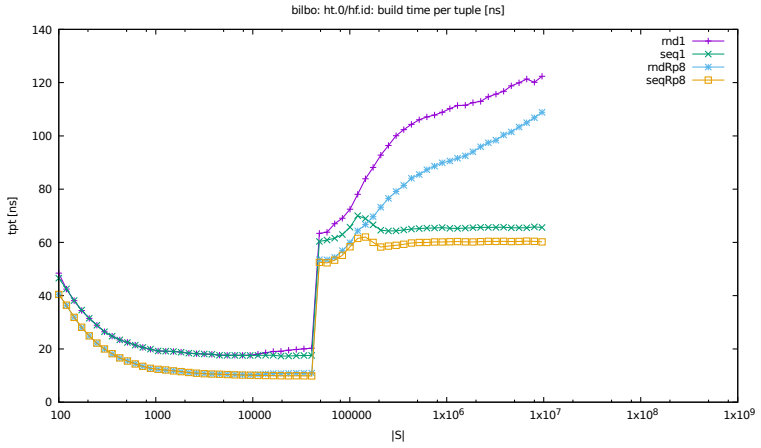


Figure 9.3: Runtime of build with and without rolling prefetch for XU-4

called AMAC.

9.3.8 Asynchronous Memory Access Chaining (AMAC)

The main idea of AMAC is to keep the state of memory accesses and executions within a small array. This array implements are ring buffer similar to rolling prefetching. However, every entry contains a complete description of the state. For the probe phase this state descriptor looks as follows:

```
struct state_t {
    uint64_t  idx;    // index/rid of current input element
    uint64_t  key;    // key of the current input element (cmp. bun_t)
    uint64_t  pload; // payload of the current input element (cmp. bun_t)
    node_t*   ptr;    // either to a hash directory entry or to a collision chain element
    int32_t   stage; // handle directory entry or collision chain element
};
```

With this state descriptor, we can modify the code for a probe such that we can emit timely prefetches.

Before we give the pseudocode for the `probe` procedure, let us clarify some details. First, it is assumed that the structure of hash directory entry equals the structure of the collision chain elements. Thus, they can be handled each by the same code fragment. Consequently, we need to distinguish two states:

1. hashing and prefetching
2. access and comparison and conditionally prefetching the next entry

In the code fragments of both cases, the state will be updated accordingly. The following gives the pseudocode for `probe`:

```
void probe(bun_t* input, uint64_t N, hashtable_t& ht, bun_t* out) {
    state_t s[SIZE]; // ring buffer of states
    int32_t k; // index into ring buffer of states
    int32_t i; // index into input array
    /* prologue: omitted here */
    while(i < N) {
        k = (k == (SIZE - 1) ? 0 : k);
        if(1 == s[k].stage) {
            entry_t* n = s[k].ptr; // collision chain entry
            if(n→key == s[k].key) {
                /* handle match: omitted here */
                s[k].stage = 0; // assume key, otherwise no 'else'
            } else if (s->next) {
                prefetch(n→next);
                s[k].ptr = n→next;
            } else {
                /* initialize new lookup (Code 0) */
            }
        } else if (0 == s[k].stage) {
            /* Code 0: hash input key, calculate bucket address */
            uint64_t h = HASH(input[i].key);
            bucket_t* ptr = &ht[h];
            prefetch(ptr);
            /* update state */
            s[k].idx = ++i;
            s[k].key = input[i].key;
            s[k].ptr = ptr;
            s[k].stage = 1;
            /* optionally: prefetch payload to emit result */
            s[k].pload = input[i].pload;
        }
        ++k;
    }
    /* epilogue: omitted here */
}
```

Note that AMAC introduces likely branch mispredictions when interpreting the status.

Exercises:

1. implement the probe phase with rolling prefetching.
2. discuss the role of miss status handling registers (MSHR)
3. work out the details of AMAC probe
4. work out the details of AMAC build
5. introduce multiple state buffers to avoid branch mispredictions

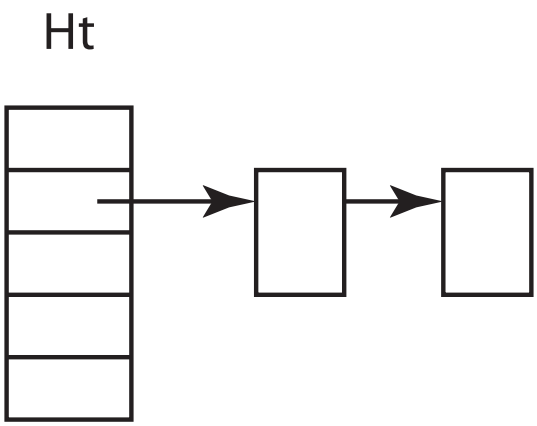


Figure 9.4: Simple partitioning/clustering

9.4 Partitioning

Partitioning is often applied to partition a big input into smaller parts each fitting some cache. The idea is to reduce random memory accesses resulting in many cache misses. Typically, a goal of partitioning is to store items in the partitions in close neighborhood, i.e., *clustered*.

We start with a very simple partitioning/clustering algorithm filling a structure as in Fig. 9.4. A simple hashtable is used to point to the partitions which are allocated in chunks and possibly chained.

A typical code fragment could look as follows [181]:

```
#define HASH(v) ((v >> 21) XOR (v >> 13) XOR (v >> 7) XOR v)
typdef struct { int v1, v2; } bun_t;
radix_cluster(bun_t* dst[2^D],        // output buffer begin
              bun_t* dst_end[2^D],    // output buffer end
              bun_t* rel,             // input relation begin
              bun_t* rel_end,         // input relation end
              int    R,               // radix bits (position)
              int    D) {             // #radix bits (depth)
  int idx, M = (1 << D) - 1;
  for(bun_t* cur = rel; cur < rel_end; ++cur) {
    idx = ((*HF)(cur→v2) >> R) & M; // use HASH
    memcpy(dst[idx], cur, sizeof(bun_t)); // use assignment
    if(++dst[idx] ≥ dst_end[idx])
      REALLOC(dst[idx], dst_end[idx]);
  }
}
```

where REALLOC can have several meanings:

- add a new chunk to the chain
- perform a real `realloc`

no	012
50	010
32	000
72	000
68	001
1	100
59	110
66	010
72	000
36	001
45	101

$\Longrightarrow$ $2msb$

no	0**12**
32	000
72	000
1	100
72	000
50	010
59	110
66	010
68	001
36	001
45	101

$\Longrightarrow$ $1lsb$

no	**012**
32	000
72	000
72	000
1	100
50	010
66	010
59	110
68	001
36	001
45	101

Figure 9.5: 2-pass radix partitioning

The code contains two comments concerning some optimization potential. Instead of a function call to the hash function and memcpy, specialized code can be used for some types.

The code is a little more complicated than expected because it can be used to *radix-partition* an input relation in more than one pass. To see why this might be useful, consider the cases where

- 2^D pointers are larger than Ld1/2/3, TLB1/2.
- 2^D exceeds the number of TLB1/2 entries.

Since the algorithm if used as a one-pass algorithm ignores cache properties, it is sometimes called *cache-oblivious* or, since it works best if everything fits into the cache *in-cache* [203].

A two pass radix-partitioning is illustrated in Fig. 9.5. Adding more passes during partitioning is a general technique to reduce the cache/TLB misses.

One problem with the approach of having chained output chunks is a possible underutilization of memory as some chunks maybe partially filled.

In order to produce a dense output, some partitioning algorithms produce a histogram first. In the following f is the function used for partitioning. It can be a combination of a hash function and selecting some appropriate bits as in the `radix_cluster` algorithm. The purpose of the histogram is to count the number of entries having a certain value under f. Let T be some input table with an attribute `key`.

build_hist(H, T) {
  H = {0};
  for(int i = 0; i < $|T|$; ++i) H[$f(T[i]$.key)]++;
}

Taking the prefix sums of the histogram, we can calculate the start offsets of each partition. Let H be some input histogram and O the offset array to be produced.

```
offset_start(O, H) {
    int off = 0;
    for(int i = 0; i < H.size(); ++i) {
        O[i] = off;
        off += H[i];
    }
}
```

The actual cache-oblivious partitioning then looks as follows:

```
part0(S, O, T) {
    for(int i = 0; i < |T|; ++i) {
        t = T[i]; // input tuple t
        off = O[f(t.key)] + +; // output index
        S[off] = t; // write output tuple to partition P
    }
}
```

Again, if the offset array and the number of output partitions are large, there are the usual problems with caches and TLBs.

These can be avoided by using multiple passes. In this case, an in-place partitioning would be helpful. Therefore, we need to calculate the offsets of the ends of each partition:

```
offset_end(O, H) {
    int off = 0;
    for(int i = 0; i < H.size(); ++i) {
        off += H[i];
        O[i] = off;
    }
}
```

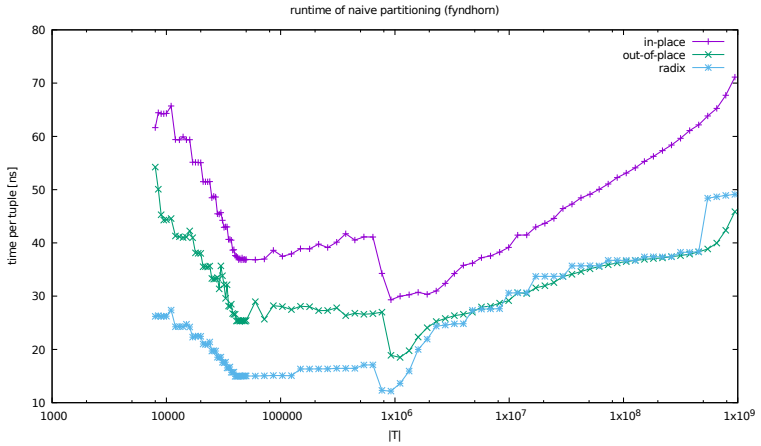
If T is now the input and output table, O the offset array produced by `offset_end`, H is the histogram, and P is the number of partitions, *in-place partioning* works as follows [203]:

```
part_in_place(O, T, H, P) {
    int off = 0, p = 0, i = 0;
    while(0 == H[p]) ++p; // skip empty partitions
    do {
        t = T[i];
        do {
            p = f(t.key);     // determine partition
            off = − − O[p];   // determine/update offset
            swap(T[off],t);   // swap current tuple with contents of destination
        } while(off != i);    // until we found something for the original place
        do {
            i += H[p++];   // skip empty/processed partitions
        } while((p < P) && (i == O[p]));
```

```
    } while(p < P);
}
```

The algorithms works by finding the correct partition for some item t. And this item is then swapped with the item t' stored at this position to allow t to be stored there. Next, we need to find a place for t'. This is done the same way until we find an item that goes to the original place $T[i]$. This closes one swap cycle.

The following figure shows the runtime of the out-of-place, in-place, and radix-cluster algorithms presented above. The x-axis contains cardinality of the input relation. The number of partitions is chosen such that a partition fits into the L1 cache. The experiment was run on a Intel Xeon E5-2620 v4 (2.10 GHz).



In the next step, we improve this algorithm by introducing a software managed buffer. The size of the buffer will be one cache line. We assume that L entries fit into a buffer. We use the last entry `buffer[L-1]` to store the offset so that we don't need an extra offset array. P denotes the number of partitions, H the histogram, S the output, T the input. The following algorithm works out-of-place [203]:

```
partition_smb(S, T, H, P) {
    int off = 0;
    for(int p = 0; p < P; ++p) {
        buffer[p][L-1] = off; // store offset of partition p
        off += H[p];
    }
    for(int i = 0; i < T.size(); ++i) {
        t = T[i]; // get next tuple
        p = f(t.key); // determine its partition
        off = buffer[p][L-1]++; // its offset
        buffer[p][off mod L] = t; // store t in buffer
        if((off mod L) == (L - 1)) {
            // flush buffer to S[off] using streaming store
```

```
            buffer[p][L-1] = off + 1;
        }
    }
}
```

An in place version of this algorithm is also presented in [203].

9.5 Sort Operator

9.6 Grouping and Aggregation

- hash-based, sort-based
- horizontal SIMD

Chapter 10

Indexing

Most index structure use pointers. The problem of storage layouts for pointered data structures while miminizing cache and TLB misses is discussed in [64].

10.1 T-Tree

Lehman and Carey [165] describe the T-Tree as an index structure useful for in-memory databases and show its superiority compared to the AVL-Tree and the traditional B-tree.

10.2 Cache Conscious B$^+$-Tree

The cache-conscious B$^+$-Tree (CSB$^+$-Tree) was developed by Rao and Ross [220]. It comes in several different flavors. The idea is to reduce the amount of memory used by pointers to child nodes. Instead of $k+1$ child pointers for k keys, the CSB$^+$-Tree stores only one or a few child points. We discuss only the former case. One child pointer suffices if successive child nodes are stored consecutively in memory. Then, a pointer to the first child is sufficient because subsequent sibling nodes can be accessed by offsets. This saving of child pointers results in better performance if key sizes are small. For large keys (much larger than a pointer size), the saving becomes less prominent.

In its simplest variant (full CSB$^+$-Tree), there is always (!) space allocated for the maximum number of child nodes. Further, Rao and Ross propose to use much smaller node sizes than for regular B$^+$-Trees: one cache line or a few thereof.

As usual, a CSB$^+$-Tree of order d contains k keys with $d \leq k \leq 2d$. The layout of an inner node of the size of a 64-byte cache line can be described as follows:

```
struct csb_node_inner_t {
    csb_node_inner_t*   _childs;          // 8 Bytes
    uint16_t            _leaf_indicator;  // 2 Bytes
    uint16_t            _no_keys;         // 2 Bytes
    uint32_t            _unused;          // 4 Byte
```

```
    int32_t          _keys[12];      // 2d keys, d = 6
}
```

All child nodes of an inner node are contained in one *node group* allocated together. There are different choices possible:

1. whenever there is an inner node, all $2d + 1$ child nodes are allocated in one node group.
   This results in the full CSB$^+$-Tree.

2. only those nodes which are actually present are allocated

3. more than one pointer (say 2 or 3) are used in inner nodes and a node group is split into *node segments*.
   This results in the segmented CSB$^+$-Tree.

Memory management is simpler in the first case and it is faster if the update/search ratio increases. However, some space is wasted.

Rao and Ross like leaf nodes to be chained (for the same reason as for B$^+$-Trees). Thus, any leaf node could be equipped with two pointers. However, since addresses to sibling leaf nodes can easily be calculated if they lie in the same node-group, only the first and the last leaf node in a node group need one pointer. The rest of the nodes is filled with key-tid-pairs. Thus, there are two different leaf page layouts.

The bulkload, insert, delete, search algorithms remain mainly the same as for regular B$^+$-Trees except that memory management (performed in node groups) becomes a little more complicated and some more copying is necessary.

10.3 Skip Lists

10.4 ART

Leis, Kemper, Neumann developed the adaptive radix tree (ART) with different node layouts [170].

Exercise:

1. Many index structures use ordered arrays of keys and apply binary search to find a match. Discuss a straight-forward implementation of binary search and try to find improvements.

Chapter 11

Boolean Expressions

Read [145, 146, 147]

Chapter 12

Cardinality Estimation

Many techniques:

1. histograms
2. sampling
3. sketches
   - to estimate the number of distinct value
   - to estimate the self-join and join sizes
4. compression using DCT, wavelets, etc.

For histograms and sketches to estimate the number of distinct values, see 'Building Query Optimizers' or [74]. The latter contains an overview of many different estmation techniques.

12.1 Sketches for Joins

12.1.1 Tug-Of-War (AGMS)

Self-Join Size

The Tug-Of-War algorithm was presented in [13, 14]. Although both papers contained more sketches, this sketch became later known as the AGMS sketch. We first start out by giving the AGMS sketch to estimate the self-join size of some relation, which coinsides with the second frequency moment. Then, we extend this procedure to produce join size estimates.

Let $\vec{f} = (f_1, \ldots, f_n)$ be a frequency vector for values $v_1, \ldots, v_n$. Define *frequency moments*

$$F_k := \sum_{i=1}^{n} f_i^k$$

Then,

- F_0: number of occurring distinct values $\leq n$

- F_1: cardinality (sum of the frequencies)
- F_2: sum of square of frequencies: selfjoin size

Let ζ_i random variables in $\{-1, +1\}$. Then $E(\zeta_i) = 0$ for all i.
Define a random variable

$$Z = \sum_{i=1}^{n} \zeta_i f_i$$

The trick of using this kind of random variables defined in $\{-1, +1\}$ was already used in earlier papers on sketches (e.g. [58]).

Using the random variable Z, define the random variable X as

$$X = Z^2.$$

We show that

$$E(X) = F_2$$

Proof: With $E(\zeta_i) = 0$ and two-way independence we have

$$\begin{aligned} E(X) &= E(Z^2) \\ &= E\big((\sum_{i=1}^{n} \zeta_i f_i)^2\big) \\ &= \sum_{i=1}^{n} f_i^2 E(\zeta_i^2) + 2 \sum_{1 \le i < j \le n} f_i f_j E(\zeta_i) E(\zeta_j) \\ &= \sum_{i=1}^{n} f_i^2 \\ &= F_2 \end{aligned}$$

□

Next, we show that

$$\mathrm{Var}(X) \le 2F_2^2$$

Proof: Similar to the above, using 4-way independence it follows that

$$E(X^2) = \sum_{i=1}^{n} f_i^4 + 6 \sum_{1 \le i < j \le n} f_i^2 f_j^2$$

(Note: $\binom{4}{2} = 6$, and due to 4-way independence we have $E(\zeta_{i_1}\zeta_{i_2}\zeta_{i_3}\zeta_{i_4}) = E(\zeta_{i_1})E(\zeta_{i_2})E(\zeta_{i_3})E(\zeta_{i_4})$) It follows that

$$\begin{aligned} \mathrm{Var}(X) &= E(X^2) - E(X)^2 \\ &= 4 \sum_{1 \le i < j \le n} f_i^2 f_j^2 \\ &\le 2F_2^2 \end{aligned}$$

□

Alghough the variance is bounded, it is quite high. Thus, we take many counters and calculate their averages and then the median of the averages to make the estimate more precise and reliable. Let s_1, s_2 be positive integers. Let $\zeta_{i,j}$ be 4-universal hash functions.

1. define $s := s_1 s_2$ random variables

$$Z_{i,j} = \sum_{v=1}^{n} \zeta_{i,j}(v) f_v$$

for $1 \leq i \leq s_1$ and $1 \leq j \leq s_2$. and another s random variables

$$X_{i,j} = Z_{i,j}^2$$

2. define s_1 random variables

$$Y_i = (1/s_2) \sum_{j=1}^{s_2} X_{i,j}$$

for $1 \leq i \leq s_1$.

3. define a random variable Z containing the median of the Y_j.

Then Z is the estimate of F_2.

Increasing s_1 increases precision, while increasing s_2 increases confidence. So far, this prodedure works nicely for self-join estimates. We have the following theorem [13, 14]:

Theorem 1 *Let R be a relation with a frequency vector f, numbers s_1, s_2, and the random variable Y as above. Then*

$$Prob\left(\frac{|Y - SJ(R)|}{SJ(R)} \leq \frac{4}{\sqrt{s_1}}\right) \geq 1 - 2^{-s_2/2}$$

where $SJ(R)$ denotes the selfjoin size of R.

Join Size

We now extend the above AGMS sketch for selfjoin size estimation to estimate join sizes. Instead of one counter for one relation, we now need two counters, one for each relation. Assume the relations are R_1 and R_2. Then, the corresponding counters Z^1 and Z^2 are defined as

$$\begin{aligned} Z^1 &:= \sum_{i=1}^{n} \zeta_i f_i \\ Z^2 &:= \sum_{i=1}^{n} \zeta_i g_i \end{aligned}$$

where f_i is the frequency of the value of value i in R_1 and g_i is the frequency of the value i in R_2. Then, the estimate is

$$Z := Z^1 * Z^2.$$

Then

$$\begin{aligned} E(Z) &= |R_1 \bowtie R_2| \\ \mathrm{Var}(Z) &\leq 2\mathrm{SJ}(R_1)\mathrm{SJ}(R_2) \end{aligned}$$

where $\mathrm{JS}(R_i)$ is the self-join size of R_i.

[Note: The AGMS sketch, like all sketches in this section, require a 4-universal family of hash functions. Those by Dietzfelbinger introduced earlier can be used.]

For the implementation, we again average and median. We need

- two numbers s_1 and s_2
- define $s = s_1 s_2$
- the counters: two vectors of size s: $Z[2]$

The insert procedure of AGMS (Tug-of-War):

```
insert(const int aVal, const int aCount, const uint aRelNo) {
    for(uint i = 0; i < s(); ++i) {
        Z[aRelNo][i] += hash(aVal, i) * aCount;
    }
}
```

where `hash`(v,i) applies the i-th hash function to the value v.

The estimate procedure:

```
double
estimate(const uint aRelNo1, const uint aRelNo2) const {
    double_vt v(s2());
    // 1. calculate averages
    uint k = 0;
    for(uint j = 0; j < s2(); ++j) {
        v[j] = 0;
        for(uint i = 0; i < s1(); ++i) {
            v[j] += Z[aRelNo1][k] * Z[aRelNo2][k];
            ++k;
        }
        v[j] /= s1();
    }
    // calculate median by sorting
    std::sort(v.begin(), v.end());
    if(0 == (v.size() & 0x1)) {
        return (v[v.size() / 2 - 1] + v[v.size() / 2]) / 2 ;
    }
    return v[v.size() / 2];
}
```

The major problem with the AGMS sketch (tug-of-war) is its insertion time: for every tuple every counter has to be updated. This is remedied by the FastAGMS sketch discussed next.

12.1.2 FastAGMS

FastAGMS sketches where proposed in [73] as a refinement of the Tug-Of-War (AGMS) sketch [13, 14]. Instead of updating s counters, only s_2 counters are updated.

In the FastAGMS sketch, s_2 sketch vectors exist for both a relation and each sketch vector Z_i consists of s_1 counters. Let $U = \{v_1, \ldots, v_n\}$ be the domain of the join attribute. Next, we need a family of hash functions $h_{1,j}$ $(1 \leq j \leq s_2)$ to map values to counters in the sketch vector (here treated as a hash table). As in the AGMS sketch, we need a familiy of hash functions $h_{2,j}$ to map values to ± 1.

- $h_{1,j} : U \rightarrow \{1, \ldots, s_1\}$ to map a value to a counter
- $h_{2,j} : U \rightarrow \{-1, +1\}$ as before

Upon an insertion or deletion with count c, we update only s_2 counters:

```
insert(const int v, const int aCount, const uint aRelNo)
    for(uint j = 0; j < s2; ++j)
        Z_aRelNo[j * s1 + h_1,j(v)] += aCount * h_2,j(v);
```

The estimation procedure now takes the median of s_2 dot products of the frequency vectors

$$\text{estimate}(a, b) := \text{median}_{1 \leq j \leq s_2} Z_a[j] \cdot Z_b[j]$$

for relation numbers a and b in $\{0, 1\}$. In pseudocode:

```
double
estimate(const uint aRelNo1, const uint aRelNo2) const
    double_vt v(s2());
    uint k = 0;
    for(uint j = 0; j < s2(); ++j)
        v[j] = 0;
        for(uint i = 0; i < s1(); ++i)
            v[j] += Z_aRelNo1[k] * Z_aRelNo2[k];
            ++k;
    std::sort(v.begin(), v.end());
    if(0 == (v.size() & 0x1))
        return (v[v.size() / 2 - 1] + v[v.size() / 2]) / 2 ;
    return v[v.size() / 2];
```

12.1.3 FastCount

FastCount sketches where proposed in [249].

12.1.4 CountMin

CountMin sketches where proposed in [75].

Chapter 13

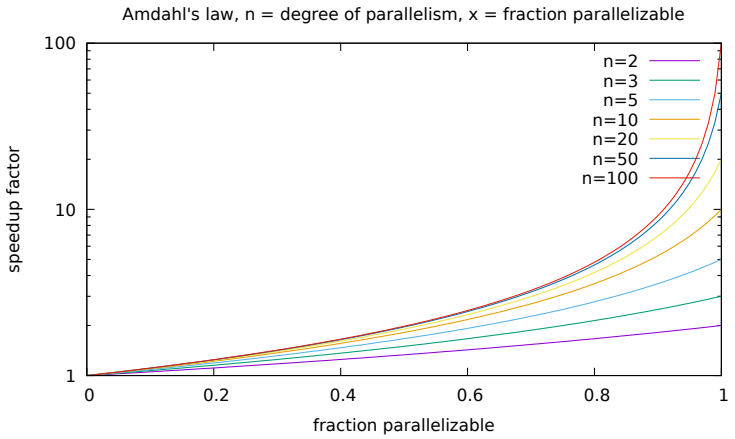
Parallelism

13.1 Amdahl's Law

Given some task t, such that a fraction x of it is parallelizable. Thus, $1 - x$ is the sequential fraction of t. For a given degree of parallelism n we can calculate the speedup factor according to *Amdahl's* law as

$$\text{speedup} = \frac{1}{1 - x + x/n}$$

Plotting this formula for different n results in:



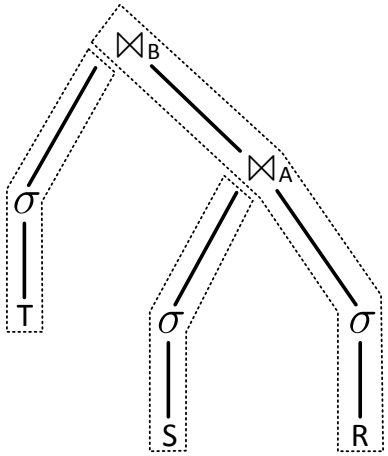
Fortunately, in the database context, we do the same task on many tuples (data parallelism).

13.2 Parallelization Constructs, Frameworks, and Libraries

- low-level: C++ threads, pthreads and synchronization primivites
- parallel patterns: parallel for, parallel reduce, fork/join parallelism
- parallel frameworks: Intel Thread Building Blocks (TBB), OpenMP, Cilk Plus

13.3 Kinds of Parallelism

Consider the following simple plan[1]:



kinds of parallelism

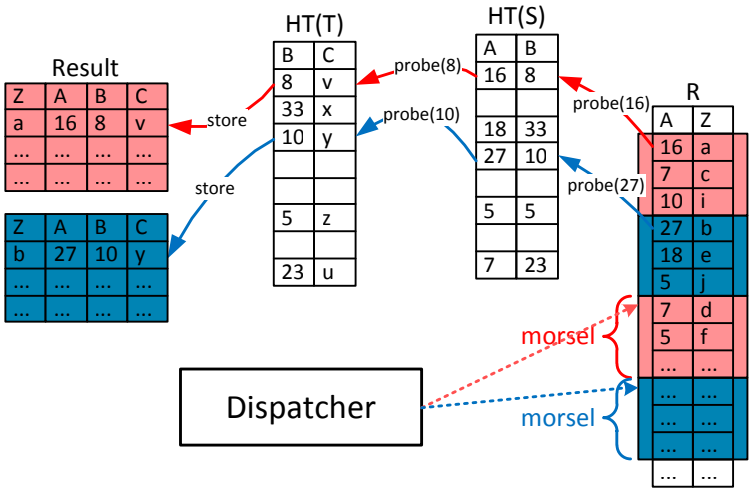
- inter-query parallelism
  - run independent queries in parallel
- intra-query parallelism:
  - partition relation and process partitions in parallel (within strands)
  - process indendent strands in parallel (*bushy parallelism*)

13.4 Morsel-Driven Parallelism

The general idea of morsel-driven parallelism [169] is the same as used in DB2 BLU, where a *morsel* is called a *stride* [90]. No matter how you name it, it is just a bunch of tuples, typically in the thousands. It is used for load-balancing by work stealing. In DB2, the number of tuples in a stride is adjusted to cache size.

We discuss the morsel-driven parallelism since the paper contains more details. Consider the plan $R \bowtie_A S \bowtie_B T$. The following picture shows the details of the last strand of the above plan:

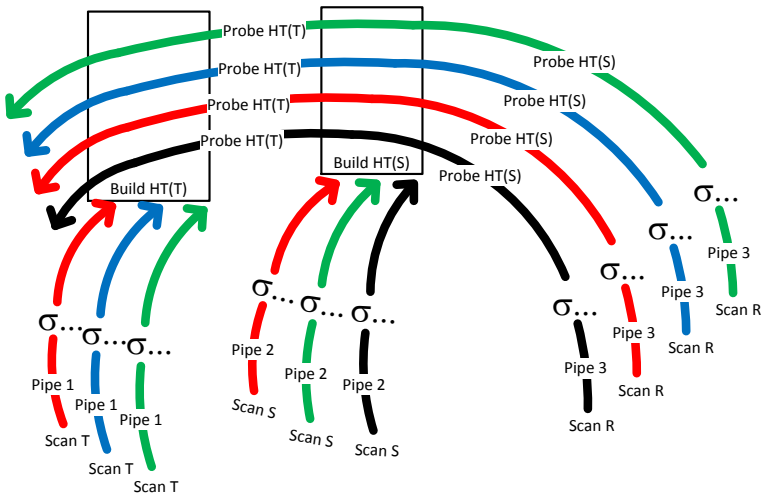
[1] Thanks to Thomas for the pictures in this chapter.



A strand is called *evaluation chain* in the context of DB2 BLU.

- relation R is partitioned into 'small' partitions called *morsels* (at least 10.000 tuples)
- each morsel is processed by some worker-thread
- the *dispatcher* determines the worker-thread
- there is one *worker-thread* for every hardware thread

A complete picture for the whole plan looks as follows:



How do we achieve NUMA-awareness?

- relation partitioned
- each partition stored at some NUMA-node
- goal: minimize traffic between NUMA-nodes

We discuss first the build-phase, then the probe phase.

Build:

- build-phase split into two phases:

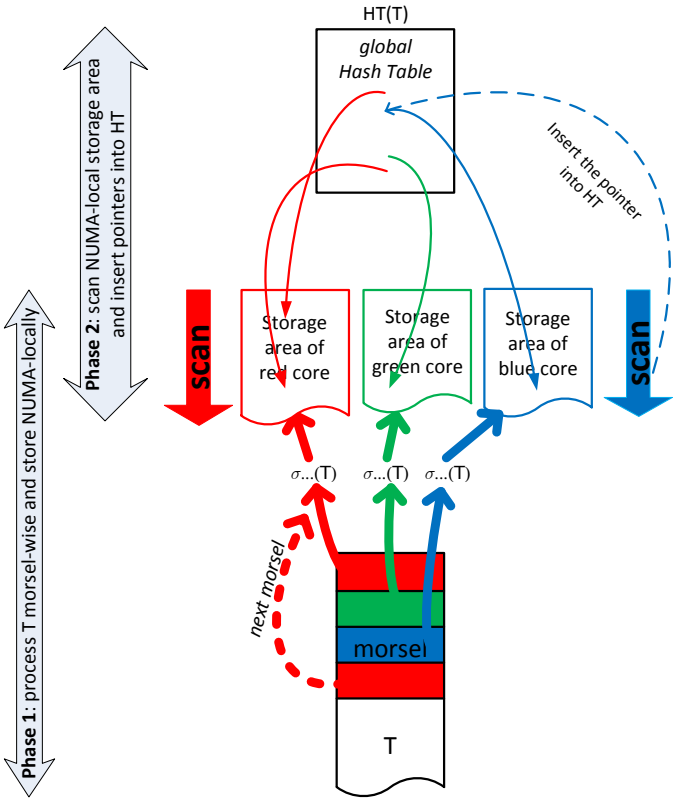
  Mat materializes the input

  HtBuild builds the hash-table

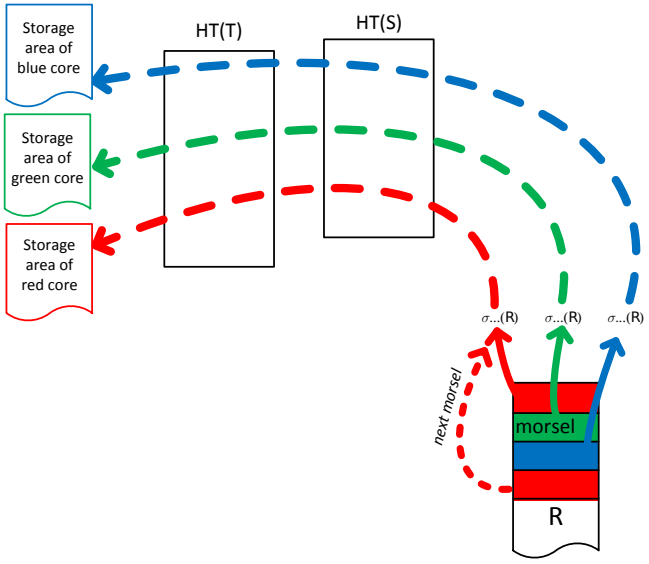
- while scanning a morsel of the input relation on a certain NUMA-node, materialization takes place on the same NUMA-node.

[Note: after materialization the exact size of the input relation is available and it can be used to allocate a hash table of perfect size.]

This is illustrated in the following figure. Colors encode worker threads confined to NUMA-nodes and memory areas belonging to NUMA-nodes.



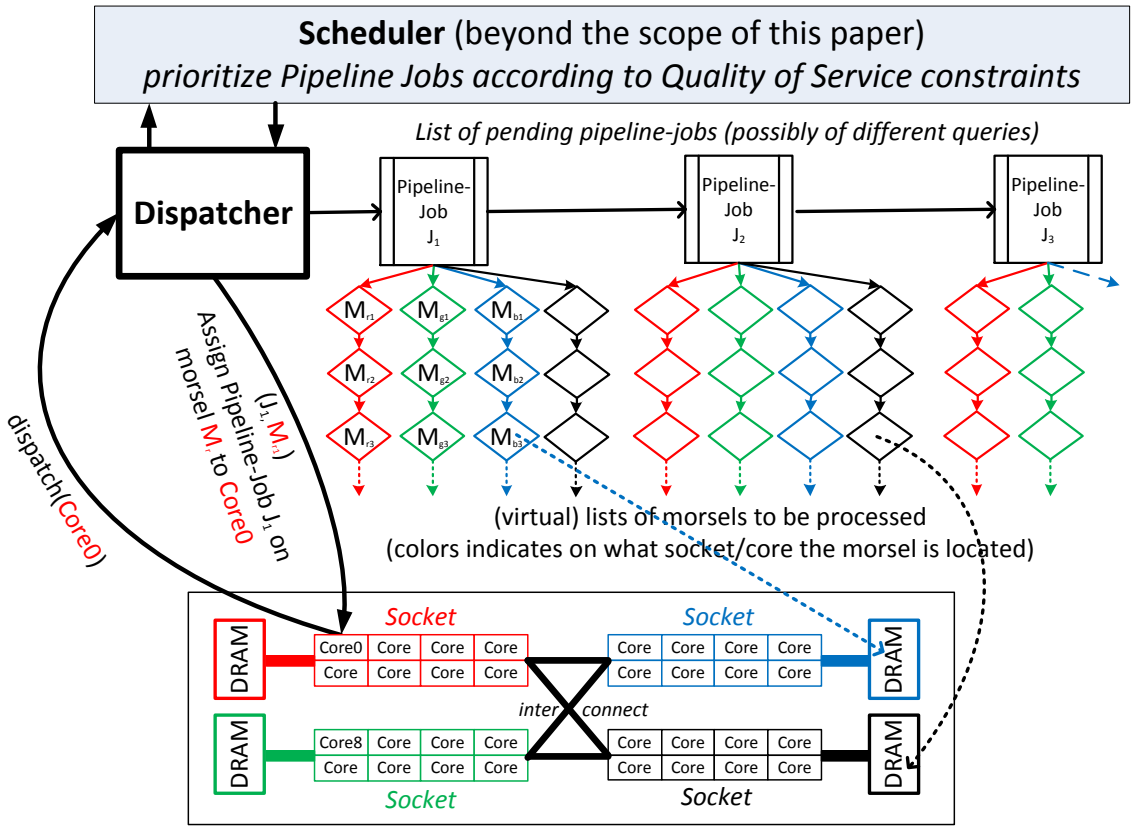
Probe:



Details:

- `QEPobject`
- *Dispatcher*
- latch-free hashtable

Dispatcher:



Remarks:

- the pipeline only contains jobs whose prerequisites are fulfilled
- the dispatcher is implemented as a latch-free datastructure
- `QEPobject` is implemented as a state-machine.
- the dispatcher code is executed by some worker thread looking for work (not as its own thread)
- the dispatcher calls `QEPobject` to generate new entries. again, this is done by a worker-thread looking for work
- although possible, Thomas stays away from bushy parallelism (reason: cache locality (discuss))
- aborting a query:
  - at any abort inducing event: mark query as aborted
  - check query after a morsel finishes
- work-stealing is supported, prefer close NUMA-nodes

13.5 Synchronization of Index Structures

Take ART as an example [171].

Chapter 14

Memory Management

Alternatives:

1. rely on malloc/free
2. have distinguished allocators
3. do it variable-sized or in chunks/pages

McAuliff, Carey, Solomon [184]:

- 4 bits per page indicates fill degree
- additional datastructure to allow fast access
  - histogram
  - cache

Chapter 15

Thread Architecture

Chapter 16

Transaction Processing

16.1 Handling updates

- in-place
- delta/staging

16.2 Lock Manager

Remember compabibility matrix for multi-granularity locking from DBSI [117]:

compatibility matrix							
	already granted						
requested	NONE	IS	IX	S	SIX	U	X
IS	+	+	+	+	+	-	-
IX	+	+	+	-	-	-	-
S	+	+	-	+	-	-	-
SIX	+	+	-	-	-	-	-
U	+	-	-	+	-	-	-
X	+	-	-	-	-	-	-

which has been extended by the deadlock-preventing U lock mode. Note the asymmetry of the U-lock.

If one transaction holds a lock and requests another one, we need the lock conversion table (used to calculate `lock_max`):

compatibility matrix							
	already granted						
requested	NONE	IS	IX	S	SIX	U	X
IS	IS	IS	IX	S	SIX	U	X
IX	IX	IX	IX	SIX	SIX	X	X
S	S	S	SIX	S	SIX	U	X
SIX	SIX	SIX	SIX	SIX	SIX	SIX	X
U	U	U	X	U	SIX	U	X
X	X	X	X	X	X	X	X

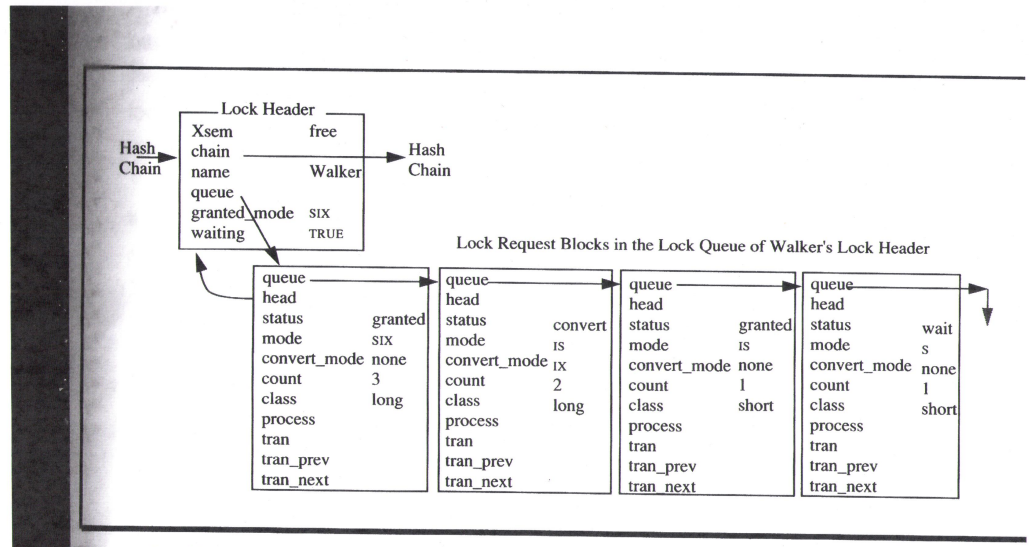


Figure 8.9: Four lock request blocks queued on a lock named Walker. Two requests are granted; one is waiting for a conversion, and one is waiting because the conversion is waiting. To simplify the display, some pointers are not diagrammed. Did the conversion arrive before the waiting S lock request?

Figure 16.1: Gray Reuter Lock Manager

16.2.1 The Gray/Reuter Lock Manager

In this section we briefly discuss the lock manager as described by Gray and Reuter [117]. Fig. 16.1 contains an overview of the data structures used.

Subsequently, we assume that a transaction control block has at least the following members:

```
struct TransCB {
    lock_request*    _locks;        // locks hold by TA
    lock_request*    _wait;         // lock TA is waiting for
    TransCB*         _cycle;        // used by deadlock detector
};
```

The interface to the lock manager can easily be described:

```
enum LOCK_REPLY { LOCK_OK,
                  LOCK_TIMEOUT,
                  LOCK_DEADLOCK,
                  LOCK_NOT_LOCKED };
LOCK_REPLY  lock(lock_name name, lock_mode mode, lock_class class, long timeout);
LOCK_REPLY  unlock(lock_name name);
```

The major components of the lock manager are:

lock hash table map data item to lock (chain), each hash directory entry contains a lock_hash struct

lock_head contains lock name, next pointer, latch, summary information about the lock queue, `lock_head`ers are pointed to by the hash directory and they are chained.

lock_request a lock points to a list of lock requests containing owner, mode, duration, etc, and a pointer to the lock header

transaction lock list for every transaction, the transaction control block holds a list of locks (see `_locks` of TransCB) held by it.

pools for efficient memory management, we have lock header free pool.

enum LOCK_MODE { ... SIX ... };

Sometimes it is helpful to know for how long a lock will be requested:

```
enum LOCK_CLASS {
    LOCK_INSTANT,       // unlock: almost directly after lock
    LOCK_SHORT,         // unlock: end of statement
    LOCK_MEDIUM,        // lock/unlock: explicit (for cursor stability)
    LOCK_LONG,          // unlock: end of transaction
    LOCK_VERY_LONG      // unlock: end of transaction, by class unlock
};

struct {
    xlatch_t        _latch; // protect collision chain
    lock_head*      _chain; // collision chain
} lock_hash[MAXHASH];

struct lock_head {
    xlatch_t        _latch;         // protect lock queue
    lock_head*      _next;          // next in collision chain
    lock_name       _name;          // name of this lock
    lock_request*   _queue;         // requests for this lock
    lock_mode       _granted_mode;  // granted group mode
    bool            _waiting;       // someone waiting?
};

enum LOCK_STATUS  { LOCK_GRANTED,
                    LOCK_CONVERTING,
                    LOCK_WAITING,
                    LOCK_DENIED
};

struct lock_request {
    lock_request*   _queue;     // pointer to next in lock queue
    lock_head*      _head;      // pointer back to head of queue
    LOCK_STATUS     _status;    // granted, waiting, ...
    LOCK_MODE       _mode;      // mode requested (and granted)
```

```
    LOCK_MODE    _convmode;   // if in convert wait, mode desired
    int          _count;      // number of times lock was locked
    LOCK_CLASS   _class;      // class in which lock is held (duration)
    PCB*         _process;    // process to wake up when lock is granted
    TransCB*     _ta_cb;      // transaction that requested/holds lock
    lock_request* _ta_prev;   // list of locks per transaction
    lock_request* _ta_next;   // list of locks per transaction
};
```

`lock` is split into several parts.
Part 1: signature and local variable declarations:

```
LOCK_REPLY // returns ok, deadlock, or timeout
lock(LOCK_NAME aName, LOCK_MODE aMode, LOCK_CLASS aClass, long aTimeout) {
    long         bucket;      //
    lock_head*   lock;        //
    lock_request* request;    // this lock request
    lock_request* last;       // queue end
    TransCb*     me = ...;    // pointer to callers TransCB
    LOCK_STATUS  lStat;       // failure reason in case of failure
    LOCK_REPLY   lRes;        // result of lock()
...
```

Part 2: find lock and is free case:

```
bucket = lockhash(name);                          // eval hash function
acquire(lock_hash[bucket]._latch);                // acquire bucket latch
lock = lock_hash[bucket]._chain;                  // get lock list
while((lock != 0) && (lock->_name != aName))      // walk lock list
    lock = lock->_next;                           // walk lock list
if (lock == NULL) {                               // lock is free case
    lock = lock_head_get(aName, aMode);           // allocate lock header
    lock->_chain = lock_hash[bucket]._chain       // list insert
    lock_hash[bucket]._chain = lock;              // list insert
    release(lock_hash[bucket]._latch);            // release bucket latch
    return LOCK_OK;                               // return ok
}
```

Part 3: lock not free, rerequest?

```
acquire(lock->_latch);                            // acquire lock latch
release(lock_hash[bucket]._latch);                // release bucket latch
for(request = lock->_queue; request != NULL; request = request->_queue) {
    if(request->_ta_cb == me)
        break;                                    // rerequest!
    last = request;                               // remember last lock in queue
}
if(request == NULL) {
    // new request, see below
```

```
} else {
    // deal with lock conversion, not handled (excercise)
}
```

Part 4: new lock request by this transaction

```
if(request == NULL) {                                    // new request
    request = lock_request_get(aLock, aMode, aClass); // allocate lock request
    last->_queue = request;                              // append lock request
    if((!lock->_waiting) && lock_compatible(aMode, lock->_granted_mode)) {
        lock->_granted_mode = lock_max(aMode, lock->_granted_mode);
        release(lock->_latch);
        return LOCK_OK;
    } else {
        lock->_waiting = true;
        request->_status = LOCK_WAITING;
        release(lock->_latch);
        wait(aTimeout);
        // after wakeup
        lStat = request->_status;
        if(lStat == LOCK_GRANTED);
            return LOCK_OK;
        if(lStat == LOCK_WAITING)
            lRes = LOCK_TIMEOUT;
        // release/free request: use unlock
        request->_class = LOCK_INSTANT; // make sure unlock will work
        unlock(request); // use unlock to release/free request
        return lRes;
    }
} else {
    // deal with lock conversion, not handled (excercise)
}
```

Remarks:

- sporadic wake-ups
- race conditions (see footnote 5 on page 475 of [117])
- observe state-machine on lock status
- some systems use bitmap of locks instead of max in `granted_mode`

Now, we come to `unlock`. Part 1 contains the signature and local variable declarations:

```
lock_reply
unlock(lock_name aName) {
```

```
    long            bucket;         // index of hash bucket
    lock_head*      lock;           // pointer to lock header block
    lock_head*      prev = NULL;    // previous (for list remove)
    lock_request*   request;        // current lock request in queue
    lock_request*   prev_request;   // prev lock request in queue
    TransCB*        me;             // callers TaCB
    lock_reply      lRes;           // return code
...
}
```

Part 2 finds the requestor's request

```
bucket = lockhash(aName);
acquire(lock_hash[bucket]._latch);
// find lock in chain
lock = lock_hash[bucket]._chain;
while((lock != NULL) && (lock->_name != aName)) {
    prev = lock;
    lock = lock->_next;
}
if(lock == NULL)
    goto B;
acquire(lock->_latch);
// find request in queue
for(request = lock->_queue; request != NULL; request = request->_queue) {
    if(request->_ta_cb == me)
        break;
    prev_request = request;
}
```

Part 3 handles the case of long locks, which are released by class and not by transaction. It also handles the case that a lock has been granted multiple times.

```
if(request->_class == LOCK_LONG ||
  request->_count > 1) {
  --request->_count;
  goto A;
}
```

Part 4 handles the case that only `me` has a request

```
if(lock->_queue == request &&
  request->_queue == NULL) {
  // remove lock from list
  if(prev == NULL) {
    lock_hash[bucket]._chain = lock->_next;
  else
    prev->_next = lock->_next;
```

```
    free(lock);
    free(request);
    goto B;
}
```

Part 5 handles the interesting case:

```
if(prev_req != NULL)
    prev_req->_queue = request->_queue; // remove request from queue
else
    lock->_queue = request->_queue;
free(request);
// recalculate group mode and wake-up waiters
lock->_waiting = false;
lock->_granted_mode = LOCK_FREE;
for(request = lock->_queue; request != NULL; request = request->_queue) {
    if(request->_status == LOCK_GRANTED)
        lock->_granted_mode = lock_max(lock->_granted_mode, request->_mode);
    else
    if(request->_status == LOCK_WAITING) {
        if(lock_compatible(request->_mode, lock->_granted_mode)) {
            request->_status = LOCK_GRANTED;
            lock->_granted_mode = lock_max(request->_mode, lock->_granted_mode);
            wakeup(request->_process);
        } else {
            lock->_waiting = true;
            break; // FIFO
        }
    } else {
        // convert waits not handled
    }
}
```

Part 6 does the latch release and return

```
    ...
    A: release(lock->_latch);
    B: release(lock_hash[bucket]._latch);
    return LOCK_OK;
}
```

Not covered: lock escalation/deescalation, deadlock detection, system startup/shutdown.

16.2.2 Starburst Lock Manager

- partition: fixed size (similar to page)
- segment: as ususal
- LCB: lock control block

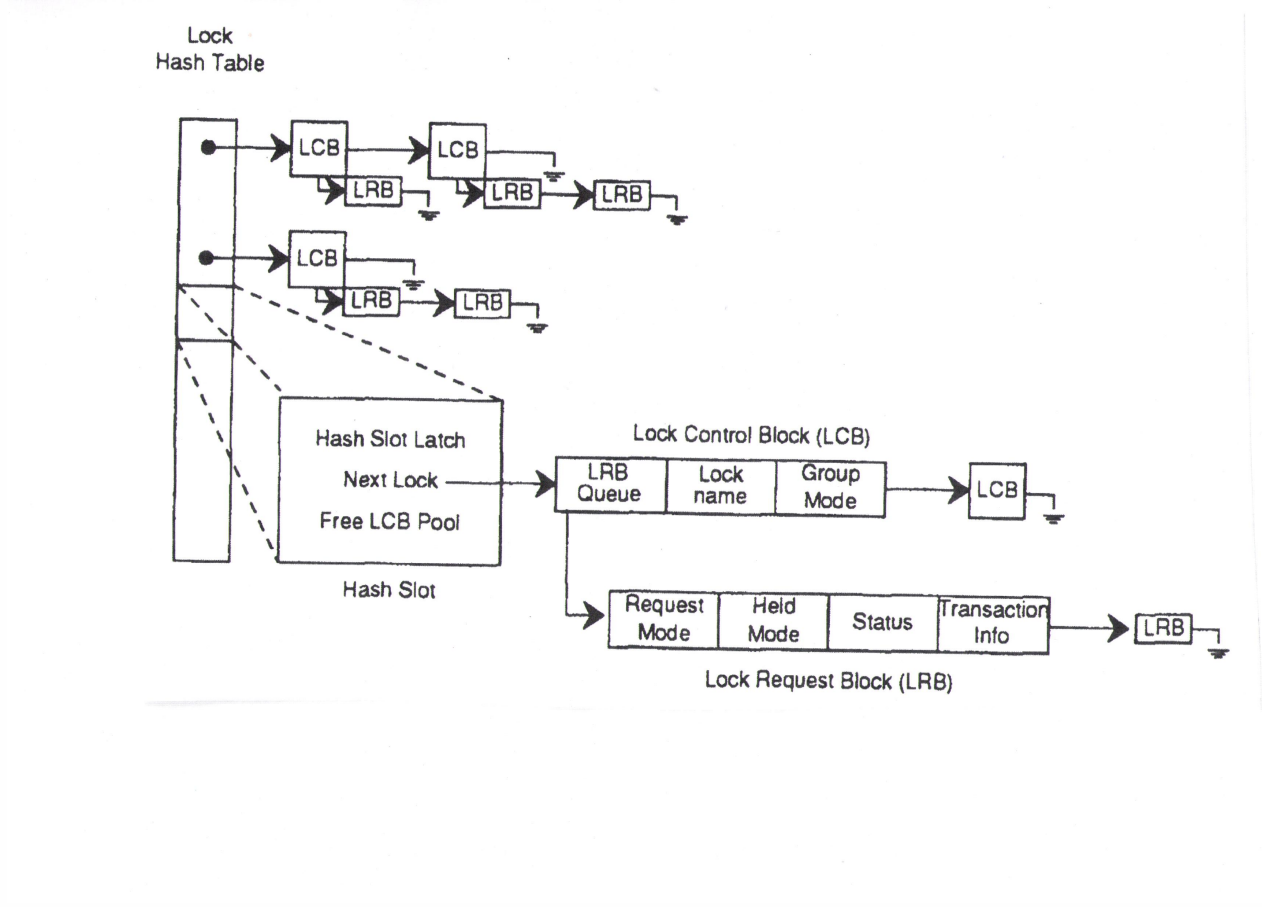


Figure 16.2: Starburst Lock Manager

- LRB: lock request block
- note: free LCB pool in slot

16.2.3 Starburst MM Lock Manager [104]

The main points of the Starburst MM Lock Manager are:

- only one latch per table protects it and all related data structures
  no extra latches for partition, index, locks
- no need for a hash table
- two levels/granularities: tables and tuples
- lock info directly attached to tables and tuples
- locking granularity flag kept in table to indicate current locking granularity
- MMM LM allows for lock escalation and deescalation (dynamically)

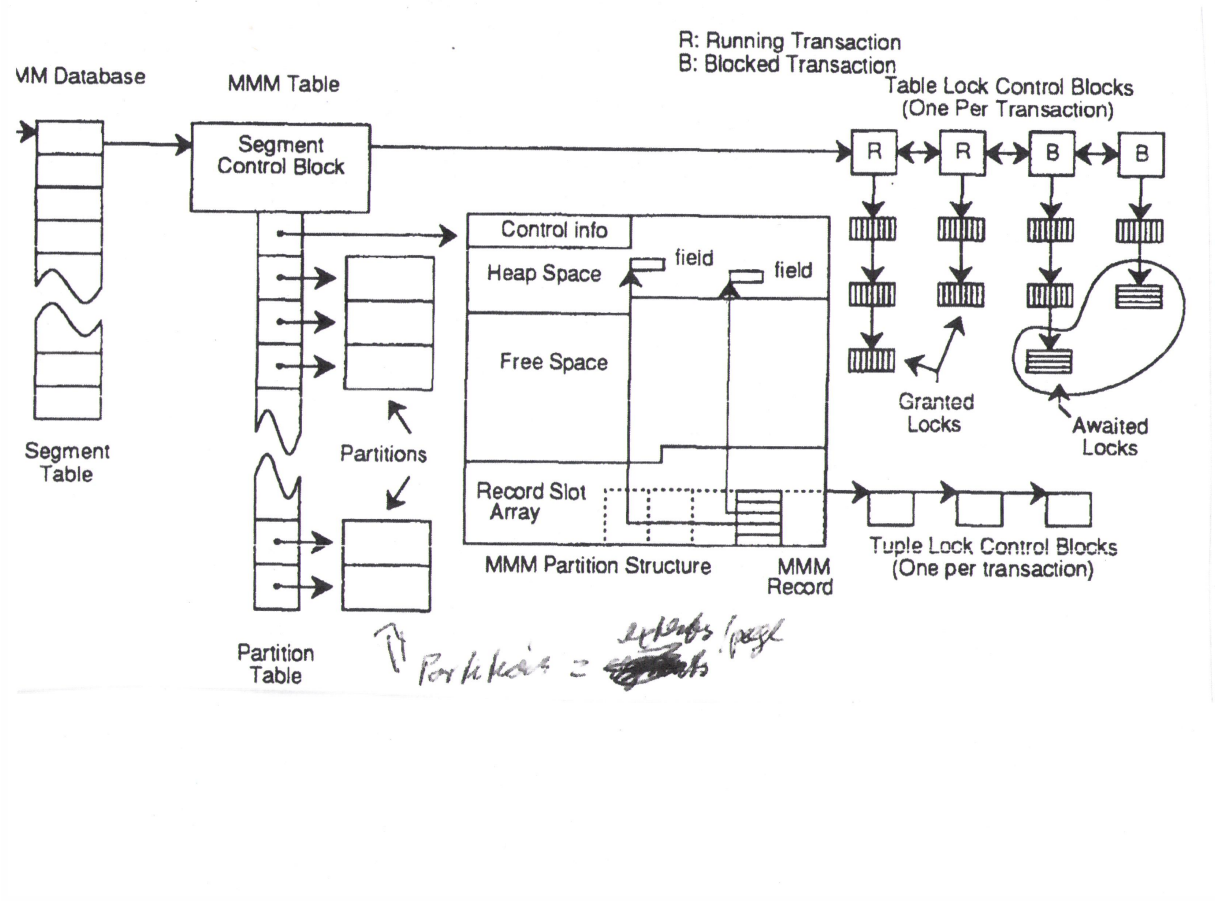


Figure 16.3: Starburst MM Lock Manager [104, 166]

- partition: fixed size (similar to page)
  slots contain real main memory pointers to tuples withing partition
- segment: variable number of partitions

In the Starburst MM Lock Manager no intention locks are used. Instead, a flag per table indicates whether locking takes place at the granularity level of a table or a tuple. If this flag indicates locking at the table granularity, all tuple level locks are automatically converted to table locks. The tuple level locks are kept in a `remembered locks` list. If the lock manager decides upon deescalation, all remembered locks are converted to real tuple locks. In case of escalation, this process is reversed.

16.2.4 Fekete Lock Manager

16.2.5 Pointers to Literature

- Agrawal, Carey, Livny: Performance of Concurrency Protocols [5]
- Blasgen, Gray, Mitoma, Price: Convoy problem [36]

16.3 Snapshot isolation

16.4 Logging

Chapter 17

The End

observations made lately:

- main memory can be too small
- main memory is expensive (compared to disk/ssd)

Thus, those parts of the data not frequently in use should reside on disk. Distinguish:

- hot data (needed)
- cold data (not needed)

hot data will be in main memory, cold data will be on disk.

Needed:

- buffer manager

And we are back in the 80s.

Acknowledgement. I thank Simone Seeger for all the beautiful pictures except those in the chapter on parallelization. They were provided by Thomas Neumann.

Appendix A

Tools and System Calls

file systems and tools for inspection

- /proc and /sys file system
- cpuid
- lscpu, lshw, lspci, lsblk, lsscsi, lsusb, lstopo
- hwlock-*
- nproc
- getconf
- hdparm
- hardinfo
- sensors (aus package lm-sensors), hddtemp

tools for measuring:

cache/memory	[264, 265], [68], [93]
X-ray	[266] [many parameters]
chi-PC, chi-T	[4] [Abel's beautiful master thesis on caches]

Tools for Performance Monitoring

- gcc -pg, gprof
- perf
- quartz for parallel apps
- MemSpy for memory system bottlenecks
- Intel Performance Counter Monitoring
- Intel Amplifier XE
- Intel Memory Latency Checker

Appendix B

Pointers to the Literature

B.1 Overview Papers/Books

Database specific:

1993 Graefe: Query Evaluation Techniques [107, 108]

1999 Härder, Rahm: Datenbanksysteme [125]

2007 Hellerstein, Stonebraker, Hamilton [130]
(they also discuss things not discussed in textbooks: e.g.: thread architecture, memory allocators)

2012 Abadi, Boncz, Harizopoulos, Idreos, Madden: Column Stores [1]

2015 Zhang, Chen, Ooi, Tan, Zhang: In-Memory Data Management [267]

2016 Färber, Kemper, Larson, Levandoski, Neumann, Pavlo [91]

2017 Ailamaki, Liarou, Tözün, Porobic, Psaroudakis [9]

Parallel Programming:

- Herlihy, Shavit: The Art of Multiprocessor Programming [133]
- Williams: C++ Concurrency in Action [261]
- Matt Kline: What every systems programmer should know about concurrency

Benchmarking/Performance:

- Gray (ed): The Benchmark Handbook for Database and Transaction Processing Systems
- Gregg: Systems Performance [118]

Computer Architecture/Assembler/ARM:

- Hennessy, Patterson: Computer Architecture: A Quantitative Approach [132]

- Patterson, Hennessy: Computer Organization and Design: ARM Edition [211]
- Pyeatt: Modern Assembly Language Programming with the ARM processor [215]

Programming:

- B. Stroustrup: The C++ Programming Language [245]
- Lipmann, Lajoie: C++ Primer (5th Edition), 2012.
- S. Myers: (More) Effective C++
- Steve McConnell: Code Complete

Programming Techniques (General):

- Roth, Sohi: prefetching for pointer-based data structures [228]
- Chilimbi, Hill, Larus: cache-consciousness for pointer-based data structures [64]
- Meyer, Sanders, Sibeyn (eds): algorithms memory hierarchies [189]

Using Compiler/Linker:

- M. Stevanovic: C and C++ Compiling

Building Compiler:

- Drachenbuch: Aho, (Lam), Sethi, Ullman: Compiler/Compilerbau (Vol1/2)
- Wait, Goos: Compiler Construction
- Wilhelm, Maurer: Compiler Design [258]
- Cooper, Torczon: Engineering a Compiler
- Srikant, Shankar (ed): The Compiler Design Handbook
- Muchnick: Advanced Compiler Design and Implementation

Operating Systems:

- Tanenbaum, Bos: Modern Operating Systems (4th Edition), 2014.
- Silberschatz, Galvin: Operating System Concepts, 2012.
- Anderson, Dahlin: Operating Systems: Principles and Practice, 2014.

B.2 Timeline (Milestones)

1984 Bratbergsengen: invention of hash-join [45]

1985 Copeland, Khoshavian: NSM, DSM [first to question NSM] [69]

1994 Shatdal, Kant, Naughton: Cache Conscious Algorithms [all techniques] [237]

1996 Boncz, Quak, Kersten: Monet DB [most influential DSM System] [41]

1997 Bratbergsengen, Norvag: partitioning on disks [46]

1998 Moerkotte: Small Materialized Aggregates [191]

1998 Helmer, Westmann, Moerkotte: Diag-Join [131]

1999 Boncz, Manegold, Kersten: bottleneck: memory [40] new version [39].

1999 Ailamaki et al.: Where does time go? [8]

2001 Rinfret et al.: Bit-slicing [224]

2001 Ailamiki et al.: PAX [7]

2002 Ross: Predicated Code [225, 226]

2003 Hankins, Patel: Data Morphing (dynamic reorganization to change storage layout) [123]

2005 Boncz et al.: MonetDB/X100 [42]

2006 Halverson et al.: Compare Disk-based Col-Store and Row-Store [121]

2009 Willhalm et al.: SIMD Scans in HANA [260]

2010 Grund et al.: HYRISE [120]

2011 Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware [197, 198]

2012 Abadi et al.: Column Oriented DBMSs (overview 2012) [1]

2013 Chasseur, Patel: Quickstep (block size matters) [59]

B.3 Storage Layouts

storage model	citations	comment
NSM	[69]	
DSM	[69]	
PDSM	[123, 213]	partially decomposed storage model = vertical partitioning
PAX	[7]	colocate (sub-)set of columns in a page
vector-blocks	[42]	similar to PAX
multi-column blocks	[3]	similar to PAX
column groups	[27]	pack groups of columns into matrix (IBM Blink)
bit packing	[259, 260]	Hana
bitslicing	[224]	
Bitweaving/H,V	[177]	here renamed to BitSliceH/V
bit packing	[95]	
padded encoding	[176]	
byte slicing	[96]	use bytes instead of bits

B.4 Memory Management

- Carey on Shore: [184]
- Wu: [263]

B.5 Hashing

Hash functions:

introductory article	[152]
universal hashing	[55, 256]
tabulation hashing	[210, 212, 249, 250, 251]
string hashing	[212, 248]

minimal perfect hash functions for strings [79]

Hash table organization:

chaining	[72, 153]
open adressing	[72, 153]
double hashing	[129] (performance several open addressing schemes)
Cuckoo	[206]
Hopscotch	[134]
Robin Hood	[56, 253]
comparison	[223]
collision	[22] cache-conscious, string hash tables

Hashing and SIMD

Cuckoo Hashing	Zukowski, Heman, Boncz [273]
Cuckoo Hashing	Ross [227]
Cuckoo Hashing	Polychroniou, Raghavan, Ross [214]

The web-page https://www.strchr.com/hash_functions?allcomments=1 contains a comparison of hash functions.

B.6 Compression

topic	citations	comment
	[15]	
	[128]	
	[140]	
	[70]	
general compr. techniques	[172]	
	[175]	
general compr.	[111, 236]	
	[43, 44]	
compr. sparse bitmaps	[192]	
	[229]	
compr. and prefetching	[78]	
	[137]	
	[18]	
	[200]	
	[222]	
row store record layout	[257]	
integration into colum store	[2]	
compression, tuned to cpu	[135]	length-lookup table for Huffman codes
	[259]	Hana
Simple8, Simple9	[16, 17]	Anh, Moffat
	[244]	Stepanov et al.
bitmap/inverted list compression	[254]	Wang et al.
decoding integers using SIMD	[173]	Lemire, Boytsov

B.7 Expression Evaluation Techniques

technique	citations	comment
interpreted/compiled tuple-wise	[167]	Lehman, Shekita, Cabrera compare AVM and com
interpreted vect. full mat.	[38]	Monet, Boncz thesis
interpreted/chunk-wise	[42]	X100
vertical SIMD	[]	
horizontal SIMD	[213]	
code generation/compilation	[219]	Rao, Pirahesh, Mohan, Lohman
code generation/compilation	[155]	Krikellas, Viglas, Cintra
code generation/compilation	[197]	Neumann
compilation vs. vectorization	[240, 241]	Sompolski, Zukowski, Boncz
code generation/compilation	[97]	Freedman, Ismert, Larson (Hekaton)
compilation (overview)	[252]	Viglas
vectorization vs. compilation	[207]	Pantela, Idreos (tiny paper)
code generation/compilation	[159]	Lang, Mühlbauer, Funke, Boncz, Neumann, Kemp
compilation/vectorization/prefetching	[188]	Menon, Mowry, Pavlo
compilation/vectorization	[89]	Kersten, Leis, Kemper, and more: comparison, bu

B.8 Indexes

technique	citations	comment
T-Tree	[165]	compares
B-Tree	[220, 124]	cache conscious B^+-tree
ART	[170, 262]	Radix Tree, different node layouts
HAT-trie	[21]	cache-conscious trie for variable length strings
B-Tree	[63]	improve B-Tree by prefetching
Bw-Tree	[138]	improved B-Tree

B.9 Physical Algebra

keyword	citations	comment
pull-model	[23]	since System R and then everywhere
push-model		Krämer, Seeger: PIPES, 2004. Some stream paper?
partitioning	[235]	Schuhknecht, Khanchandani, Dittrich: Radix Partiti
hash-join	[45]	invention of hash join
Division	[105]	4 algorithms
GRACE-Hash Join	[99, 150, 151, 196]	disk-based, tuning
Hybrid-Hash Join	[102]	disk-based
Hash Join	[179]	eliminate random I/O in hash joins
Heap-Filter Merge Join	[106]	
several joins	[34, 24, 26]	mm, comparison
mm hash join	[28]	IBM, simply the best
mm joins	[35]	
NUMA-aware join	[157]	
NUMA-aware join	[230]	
NUMA-aware join	[158]	
NUMA-aware sort-merge join	[10]	
join	[11]	numa-aware, non-inner
sort vs. hash	[110, 25, 149]	disk, mm, par
DBJ	[269, 270]	balancing in multi-processor environment
partitioned sort	[203]	radix-sort, comparison-sort
hash join	[62]	prefetching
Sorting	[136]	SIMD-based sort
	[238]	sorting large sets of strings
	[255]	parallel radix sort
Intersection	[234]	SIMD-based intersection
hash join	[61]	Inspector Join
hash join	[100, 101]	parallel
hash join	[221]	parallel
hash join	[40, 182]	radix join
grouping/aggregation	[42]	X100
grouping/aggregation	[271]	SIMD
grouping/aggregation	[195]	comparison/evaluation
grouping/aggregation	[193]	hashing is sorting
compilation templates	[155]	join, grouping/aggregation

B.10 Prefetching

group prefetching	hash join	[60, 62]
software pipelined prefetching	hash join	[60, 62]
Prefetching B^+-Tree (pB^+-Tree	index	[63]
asynchronous memory access chaining (AMAC)	hash join/grouping/index	[154]

Disadvantages:

1. group prefetching: high bursts of prefetch operations

2. software pipelined prefetching: ineffective for short pipelines

3. AMAc: buffer need structs with 5 entries, one of which is the stage of processing. if-stmt (branch!) used to execute code corresponding to the stage.

B.11 Instruction-Level Parallelism (ILP, SIMD)

SIMD overview	[127]
Bloom filters	[204]
scansel, partition, hash	[214]
Compression	[205]
selection, hash, partitioning	[271, 214]
hashing	[227]

B.12 Thread-Level Parallelism (TLP)

- Mrunal Gawade, Martin L. Kersten: Adaptive query parallelization in multi-core column stores. EDBT 2016: 353-364.

- Morsel [169]

- [164] minimize shared LLC problems

- [216] scan sharing in BLINK

- Shatdal, Naughton: Adaptive Parallel Aggregation Algorithms. SIGMOD 95.

B.13 NUMA

- Mrunal Gawade, Martin L. Kersten: NUMA obliviousness through memory mapping. DaMoN 2015: 4:1-4:7

- Li, Pandis, Mueller, Raman, Lohman: NUMA-aware algorithms: the case of data shuffling. CIDR 2013.

B.14 Cost Models

Hybrid-Hash Join	[209]	disk, cost model
memory	[181]	cost models for memory hierarchy
instructions count		
microbenchmark-based		

B.15 Code Generation

- Mehta [187]

B.16 Buffer Management

B.16.1 Buffer Manager

1966 Belady: replacement algorithms [30]

1971 Aho, Denning, Ullman: replacement algorithms [6]

1977 Lang, Wood, Fernandez: DBMS buffer and OS buffer [161]

1982 Sacco, Schkolnick: managing buffer pool [232]

1984* Effelsberg, Härder: Principles of Buffer Management [85]

1985* Chou, DeWitt: Evaluation of Buffer Management Strategies [65]

1986 Sacco, Schkolnick: buffer mgmt [233]

1987 Saccl: index access with finite buffer [231]

1989 Cornell, Yu: integration of buffer mgmt and query opt [76, 92]

1990 Dan, Dias, Yu: Skewed Data and Buffer [80]

1990 Jauhari, Carey, Livny: Priority Hints [141]

1991 Ng, Faloutsos, Sellis: Buffer Allocation [199]

1992 Jian: DFS traversal and buffering [142]

1996 Brown, Carey, Livny: Goal-Oriented Buffer Management [47]

1999* O'Neil, O'Neil, Weikum: LRU-k replacement [202]

2004 Megiddo, MOdha: adaptive replacement [186]

2012 Switakoswki, Boncz, Zukowski: Cooperative Scans, Predictive Buffer Mngmt [246]

2014* Graefe et al: Buffer for In-Memory DBMS [112]

Five-Minute Rules:

1987 Gray, Putholu [114, 115]

1997 Gray, Graefe [113]

2007 Graefe [109]

2017 Appuswamy, Gorovica-Gajic, Graefe, Ailamaki [19]

see also Gray and Shenoy: Rules of Thumb in Data Engineering [116].

B.16.2 Buffering Without Buffermanager

2013 DeBrabant et al: Anti-caching [81]

2013 Larson et al: Siberia [12, 86, 174]

2015 Zhang, Chen, Ooi, Wong: Anti-Caching for bit data [268]

B.17 Recovery/Checkpointing

1993 JaSiSu93

1997 PaBi97

2009 LeKiWoChKi09 (FLASH)

B.18 Storage Manager

1976 System R [23]

1985 Wisconsin Storage Manager ChDeKl85 [66]

1989 Exodus storage manager CaDeGrHaRiScShVa88,CaDe87,CaDeVa88,CaWiFrGrMuRiSh86,CaDeGr
53, 54, 87, 50, 52]

1994 Shore storage manager CaDeFrHaMcNaScSoTaTsWhZw94 [49]

1994 EOS storage manager BiPa94 [31]

1995 BeSS storage manager BiPa95,BiPa96 [32, 33]

1997 Dali main memory storage manager BoLiRaSiSeSu97,JaLiRaSiSu94 [37,
139]

1998 Xmax main memory storage manager ChPaPa97 [57]

1989 Datablitz main memory storage manager BaBoKhKo98 [29]

2009 Shore-MT scalable storage manager JoPaHaAiFa09 [143]

1992 Starburst main memory storage manager LeShCa92 [167]

2005 General Considerations for Lightweight Storage Manager LeApSa05 [168]

2010 HYRISE main memory hybrid storage engine GrKrPlZeCuMa10 [120]

Evaluation of Storage Managers (emphasis: main memory vs. disk-based):

1992 Lehman: Starburst MM LeShCa92 [167]

1999 Shore-MT scalable storage manager JoPaHaAiFa09 [143]

2008 Harizopoulos et al.: Looking Glass HaAbMaSt08 [126]

B.19 System Overviews

- DB2blu [90]
- SQL Server Apollo/Hekaton
- Hyper
- Hana

B.20 todo

- [46] Bratbergsengen, Norvag: partitioning on disks
- [272] instruction cache performance improvement by blocking
- [67] small overview article by Cieslewicz and Ross
- [28] IBM's memory efficient hash join

Bibliography

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2012.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 671–682, 2006.

[3] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *Proc. IEEE Conference on Data Engineering*, pages 466–475, 2007.

[4] A. Abel. Measurement-based inference of the cache hierarchy. Master Thesis, U. Saarland, 2012.

[5] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. on Database Systems*, 12(4):609–654, 1987.

[6] A. Aho, P. Denning, and J. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, 1971.

[7] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[8] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a modern processor: Where does time go? In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 266–277, 1999.

[9] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. *Databases on Modern Hardware*. Morgan&Claypool, 2017.

[10] M.-C. Albitiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10), 2012.

[11] M.-C. Albutiu, A. Kemper, and T. Neumann. Extending the MPSM join. In *BTW*, pages 57–72, 2013.

[12] K. Alexiou, D. Kossmann, and P. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proc. of the VLDB Endowment (PVLDB)*, 6(14):1714–1725, 2013.

[13] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput System Sciences*, 35(4):391–432, 2002.

[14] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. of Computer and System Sciences*, 58(1):137–147, 1999.

[15] P. Alsberg. Space and time savings through large database compression and dynamic restructuring. In *Proc IEEE 63,8*, Aug. 1975.

[16] N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[17] N. Anh and A. Moffat. Index compression using 64-bit words. *Software – Practice and Experience*, 40:131–147, 2010.

[18] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving key compression. Technical Report CRL 94/3, Digital Equipment Corporation, June 1994.

[19] R. Appuswamy, R. Borovica-Gajic, G. Graefe, and A. Ailamaki. The five-minute rule thirty years later and its impact on the storage hierarchy. In *ADMS*, 2017.

[20] ARM. *ARM Synchronization Primitives*, 2009.

[21] N. Askitis and R. Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *ACSC*, pages 97–105, 2007.

[22] N. Askitis and J. Zobel. Cache-conscious collision resolution in string hash tables. In *SPIRE 2005*, LNCS 3772, pages 91–102, 2005.

[23] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. Mc Jones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[24] C. Balkesen, J. Teubner, G. Alonso, and T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. Technical Report Technical Report Nr. 779, ETH Zürich, 2012.

[25] C. Balkesen, J. Teubner, G. Alonso, and T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. of the VLDB Endowment (PVLDB)*, 7(1):85–96, 2014.

[26] C. Balkesen, J. Teubner, G. Alonso, and T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. on Knowledge and Data Engineering*, 27(7):1754–1766, 2015.

[27] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreso, M.-S. Kim, O. Koeth, J.-G. Lee, T. Li, G. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012.

[28] R. Barber, G. Lohman, I. Pandis, G. Attaluri, N. Chainani, S. Lightstone, V. Raman, R. Sidle, and D. Sharpe. Memory-efficient hash joins. *Proc. of the VLDB Endowment (PVLDB)*, 8(4), 2014.

[29] J. Baulier, P. Bohannona, A. Khivesara, H. Korth, R. Rastogi, A. Silberschatz, and S. Sudarshan. The DataBlitz main-memory storage manager: Architecture, performance, and expercience. found on the internet, 1998.

[30] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[31] A. Biliris and E. Panagos. EOS user's guide. Technical Report Release 2.2, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1994.

[32] A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proc. IEEE Conference on Data Engineering*, pages 35–43, 1995.

[33] A. Biliris and E. Panagos. The BeSS object storage manager: Architecture overview. *ACM SIGMOD Record*, 25(3):53–58, 1996.

[34] J. Blakeley and N. Martin. Join index, materialized view, and hybrid hash-join: a performance analysis. In *Proc. IEEE Conference on Data Engineering*, pages 256–236, 1990.

[35] S. Blanas and J. Patel. Memory footprint matters: Efficient equi-join algorithms for main memory data processing. In *SoCC*, 2013.

[36] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[37] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, and S. Sudarshan. The architecture of the dali main-memory storage manager. *Bell Labs Technical Journal*, pages 1–46, 1997.

[38] P. Boncz. *Monet: A Next-Generation DBMS Kernel for Query Intensive Applications*. PhD thesis, Wiskunde en Informatica, 2002.

[39] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, 2008.

[40] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 54–65, 1999.

[41] P. Boncz, W. Quak, and M. Kersten. Monet and its geographical extensions: A novel appraoch to high performance GIS processing. In *EDBT*, pages 145–166, 1996.

[42] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[43] A. Bookstein and S. Klein. Construction of optimal graphs for bit-vector compression. In *SIGIR*, pages 327–342, 1990.

[44] A. Bookstein and S. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[45] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–333, 1984.

[46] K. Bratbergsengen and K. Norvag. Improved and optimized partitioning techniques in database query procesing. In *Advances in Databases, 15th British National Conference on Databases*, pages 69–83, 1997.

[47] K. Brown, M. Carey, and M. Livny. Goal-oriented buffer management revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 353–364, Montreal, Canada, Jun 1996.

[48] D. Butterstein and T. Grust. Precisoin performance surgery for PostgreSQL. *PVLDB*, 9(13), 2016.

[49] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, 1994.

[50] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 474–499. Morgan-Kaufman, 1989.

[51] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. Technical Report TR808, U Wisconsin, Maddison, 1988.

[52] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. *Kim, Lochovsky: [148]*, chapter Storage Management for Objects in EXODUS. Addison Wesley, 1990.

[53] M. Carey and D. J. DeWitt. An overview of the EXODUS project. *IEEE Database Engineering*, 10(2):47–53, Jun 1987.

[54] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–423, Chicago, Il., Jun 1988.

[55] L. Carter and M. Wegman. Universal classes of hash functions. *J. Comp. and Sys. Sci.*, 18(2):143–154, 1979.

[56] P. Celis. *Robin Hood Hashing*. PhD thesis, 1986.

[57] S. K. Cha, J. H. Park, and B. D. Park. Xmas: An extensible main-memory storage system. In *CIKM*, pages 356–362, 1997.

[58] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. Int. Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[59] C. Chasseur and J. Patel. Design and implementation of storage organizations for read-optimized main memory databases. *PVLDB*, 6(13):1474–1485, 2013.

[60] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. In *Proc. IEEE Conference on Data Engineering*, pages 116–127, 2004.

[61] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Inspector joins. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 817–828, 2005.

[62] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. *ACM Trans. on Database Systems*, 32(3):17, 2007.

[63] S. Chen, P. Gibbons, and T. Mowry. Improving index performance through prefetching. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 235–246, 2001.

[64] T. Chilimbi, M. Hill, and J. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.

[65] H.-T. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 127–141, 1985.

[66] H.-T. Chou, D. DeWitt, and A. Klug. Design and implementation of the wisconsin storage manager. *Software: Practice and Experience*, 15(10):943–962, 1985.

[67] J. Cieslewicz and K. Ross. Database optimizations for modern hardware. *Proc. of the IEEE*, 96(5), 2008.

[68] K. Cooper and J. Sandoval. Portable techniques to find effective memory hierarchy parameters. Technical Report CS TR11-06, Rice University, 2011.

[69] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–279, Austin, TX, 1985.

[70] G. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[71] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[72] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. 2nd Edition.

[73] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 13–24, 2005.

[74] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. NOW Press, 2012.

[75] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its application. *J. Algor*, 55(1):58–75, 2005.

[76] D. Cornell and P. Yu. Integration of buffer management and query optimization in relational database environments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 247–255, 1989.

[77] D. Culler, J. Singh, and A. Gupta. *Parallel Computer ARchitecture: A Hardware/Software Approach*. Morgan Kaufman, 1999.

[78] K. Curewith, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, page 257, 1993.

[79] Z. Czech, G. Havas, and B. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letter*, 43:257–264, 1992.

[80] A. Dan, D. Dias, and P. Yu. The effect of skewed data access on buffer hits and data connection in a data sharing environment. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 419–431, 1990.

[81] Justin DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anticaching: A new approach to database management system architecture. *Proc. of the VLDB Endowment (PVLDB)*, 6(14):1942–1953, 2013.

[82] C. Diaconu, C. Freeman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2013.

[83] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *STOCS*, pages 569–580, 1996.

[84] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. of Algorithms*, 25(1):19–51, 1997.

[85] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, 1984.

[86] A. Eldawy, J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *Proc. of the VLDB Endowment (PVLDB)*, 7(11):931–942, 2014.

[87] M. J. Carey et al. The architecture of the EXODUS extensible DBMS. In *Workshop on Object-Oriented Database Systems*, pages 52–65, 1986.

[88] P. Larson et. al. Enhancements to SQL server column stores. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1159–1168, 2013.

[89] T. Kersten et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. of the VLDB Endowment (PVLDB)*, 11(13):2209–2221, 2018.

[90] V. Raman et. al. DB2 with BLU acceleration: So much more than just a column store. In *Proc. of the VLDB Endowment (PVLDB)*, pages 1080–1091, 2013.

[91] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo. *Main Memory Database Systems*. NOW, 2016.

[92] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 265–274, 1989.

[93] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *TACO*, 11(4):55, 2014.

[94] F. Färber, N. May, W. Lehner, P. Große, I. Müeller, H. Rauhe, and J. Dees. The SAP HANA database: An architecture overview. *IEEE Data Engineering Bulletin*, 2012.

[95] M. Faust, M. Grund, T. Berning, D. Schwalb, and H. Plattner. Vertical bit-packing: Optimizing operations on bit-packed vectors leveraging SIMD instructions. In *DASFAA*, pages 132–145, 2014.

[96] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the envelop of main memory data processing with a new storage layout. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 31–46, 2015.

[97] C. Freedman, E. Ismert, and P. Larson. Compilation in the microsoft SQL server hekaton engine. *IEEE Data Engineering Bulletin*, 37(1):22–30, 2014.

[98] F. Funke, A. Kemper, and T. Neumann. HyPer-sonic combined transaction and query processing. *PVLDB*, 4(12), 2011.

[99] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 209–219, 1986.

[100] P. Garcia and H. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Proc. Conf. On Computing Frontiers*, pages 241–252, 2006.

[101] P. Garcia and H. Korth. Pipelined hash-join on multithreaded architectures. In *DaMoN*, page 1, 2007.

[102] R. Gerber. *Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms*. PhD thesis, UNIV of Wisconsin, Madison, 1986.

[103] G. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[104] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 533–544, 1992.

[105] G. Graefe. Relational division: Four algorithms and their performance. In *Proc. IEEE Conference on Data Engineering*, pages 94–101, 1989.

[106] G. Graefe. Heap-filter merge join: A new algorithm for joining medium-size inputs. *IEEE Trans. on Software Eng.*, 17(9):979–982, 1991.

[107] G. Graefe. Query evaluation techniques for large databases. Shortened version: [108], July 1993.

[108] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[109] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Int. Workshop on Data Management on New Hardware (DaMoN)*, 2007.

[110] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. on Knowledge and Data Eng.*, 6(6):934–944, Dec. 1994.

[111] G. Graefe and L. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, April 1991.

[112] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-memory performance for big data. *Proc. of the VLDB Endowment (PVLDB)*, 8(1):37–48, 2014.

[113] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[114] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disk accesses and the 5 byte rule for trading memory for CPU time. Technical Report TR86.1, Tandem, 1986.

[115] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 395–398, 1987.

[116] J. Gray and P. Shenoy. Rules of thumb in data engineering. Technical Report MS-TR-99-100, Microsoft, 1999.

[117] P. Gray and A. Reuter. *Transaction Processing: Concepts and Technology*. Morgan Kaufmann Publishers, San Mateo, Ca, 1993.

[118] B. Gregg. *Systems Performance*. Prentice Hall, 2014.

[119] R. Grisenthwaite. *ARM Barrier Litnus Tests and Cookbook*. ARM, 2009. PRD03-GENC-007826.

[120] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE — a main memory hybrid storage engine. *Proc. of the VLDB Endowment (PVLDB)*, 4(2):105–116, 2010.

[121] A. Halverson, J. Beckmann, J. Naughton, and D. DeWitt. A comparison of C-Store and Row-Store in a common framework. Technical report, University of Wisconsin, Madison, 2006.

[122] P. Hammarlund, A. Martinez, A. Bajwa, D. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsakar, R. Kumar, R. Osborne, R. Rajwar, R. Singhal, R. D'Sa, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2), 2014.

[123] R. Hankins and J. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 417–428, 2003.

[124] R. Hankins and J. Patel. Effect of node size on the performance of cache-concsious B^+-trees. *SIGMETRICS Perform. Eval. Rev.*, 31(1), 2003.

[125] T. Härder and E. Rahm. *Datenbanksysteme*. Springer, 1999.

[126] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 981–992, 2008.

[127] M. Hassablallah, S. Omran, and Y. Mahdy. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51(6):630–649, 2008.

[128] K. Hazboun and M. Bassiouni. A multi-group technique for data compression. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 284–292, 1982.

[129] G. Heilemann and W. Luo. How caching affects hashing. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 141–154, 2005.

[130] J. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2), 2007.

[131] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationships. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 98–109, 1998.

[132] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[133] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[134] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *DISC*, pages 350–364, 2008.

[135] A. Holloway, V. Raman, G. Swart, and D. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2007.

[136] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques*, pages 189–198, 2007.

[137] B. Iyer and D. Wilhite. Data compression support in databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 695–704, 1994.

[138] S. Sengupta J. Levandoski, D. Lomet. The bw-tree: A b-tree for new hardware platforms. In *Proc. IEEE Conference on Data Engineering*, 2013.

[139] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 48–59, 1994.

[140] M. Jakobsson. Evaluation of a hierarchical bit-vector compression technique. *Information Processing Letters*, 14(4):147–149, 1982.

[141] R. Jauhari, M. Carey, and M. Livny. Priority-hints: An algorithm for priority-based buffer management. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 708–721, 1990.

[142] B. Jian. DFS-traversing graphs in a paging environment, LRU or MRU? *Information Processing Letters*, 40(4):193–196, 1992.

[143] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proc. European Conf. on Extending Database Technology (EDBT)*, pages 24–35, 2009.

[144] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. pages 622–634, 2008.

[145] F. Kastrati and G. Moerkotte. Optimization of conjunctive predicates for main memory column stores. *Proc. of the VLDB Endowment (PVLDB)*, 9(12):1125–1136, 2016.

[146] F. Kastrati and G. Moerkotte. Optimization of disjunctive predicates for main memory column stores. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 731–744, 2017.

[147] F. Kastrati and G. Moerkotte. Generating optimal plans for boolean expressions. In *Proc. IEEE Conference on Data Engineering*, 2018.

[148] M. Kifer and E. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. on Database Systems*, 15(4):385–426, 1990.

[149] C. Kim, T. Kaldewey, V. Lee, E. Sadlar, A. Nguyen, N. Satish, J. Chugani, A. DiBlas, and P. Dubey. Sort vs. hash revisited: fast join implementation on multi-core CPUs. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1378–1389, 2009.

[150] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–266, 1989.

[151] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 210–221, 1990.

[152] G. Knott. Hashing functions. *Comp. J.*, 18(3):265–278, 1974.

[153] D. Knuth. *The Art of Computer Programming; Volume 3: Sorting and Searching*. Addison Wesley, 2000.

[154] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *VLDB Journal*, 9(4):252–263, 2015.

[155] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proc. IEEE Conference on Data Engineering*, pages 613–624, 2010.

[156] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 61–72, 2012.

[157] H. Lang, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel NUMA-aware hash joins. In *In-Memory Data Management and Analysis (IMDM)*, pages 3–14, 2015.

[158] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *IMDM*, pages 3–14, 2015.

[159] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 311–326, 2016.

[160] T. Lang, E. Fernandez, and R. Summers. A system architecture for compile-time actions in databases. In *Proc. ACM Annual Converence*, pages 11–15, 1977.

[161] T. Lang, C. Wood, and I. Fernandez. Database buffer paging in virtual storage systems. *ACM Trans. on Database Systems*, 2(4):339–351, 1977.

[162] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. Patel, and M. Zwillling. High-performance concurrency control mechanisms for main-memory databases. *Proc. of the VLDB Endowment (PVLDB)*, pages 298–309, 2011.

[163] P.-A. Larson, C. Clinciu, E. Hanson, A. Oks, S. Price, S. Rangajaran, A. Surna, and Q. Zhou. SQL server column store indexes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1177–1184, 2011.

[164] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 373–384, 2009.

[165] T. Lehman and M. Carey. A study of index structures for main memory database management systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 294–303, 1986.

[166] T. Lehman and V. Gottemukkala. The design ande performance evaluation of a lock manger for a memory-resident database system. Technical Report, 1994.

[167] T. Lehman, E. Shekita, and L.-F. Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Trans. on Knowledge and Data Engineering*, 4(6):555–566, 1992.

[168] T. Leich, S. Apel, and G. Saake. Using step-wise refinement to build a flexible lightweight storage manager. In *ABDIS*, 2005.

[169] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 743–754, 2014.

[170] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[171] V. Leis, F. Schreiber, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Int. Workshop on Data Management on New Hardware (DaMoN)*, 2016.

[172] D. Lelewer and D. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[173] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software – Practice and Experience*, 45:1–29, 2015.

[174] J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *Proc. IEEE Conference on Data Engineering*, pages 26–37, 2013.

[175] J. Li, D. Rotem, and H. Wong. A new compression method with fast searching on large data bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, page 311, Brighton, England, 1987.

[176] Y. Li, C. Chasseur, and J. Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2015.

[177] Y. Li and J. Patel. BitWeaving: Fast scans for main memory data processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 289–300, 2013.

[178] Y. Li and J. Patel. BitWeaving: Fast scans for main memory data processing. Technical report, U. Wisc, 2013.

[179] M.-L. Lo and C. Ravishankar. Towards eliminating random I/O in hash joins. In *Proc. IEEE Conference on Data Engineering*, pages 422–429, 1996.

[180] R. Lorie and B. Wade. The compilation of a high level data langauge. Technical report, IBM Research Laboratory, San Jose, 1979.

[181] S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. Technical report, CWI Amsterdam, 2002.

[182] S. Manegold, P. Boncz, and M. Kersten. Optimizing main memory join for on modern hardware. *IEEE Trans. on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[183] S. Manegold, M. Kersten, and P. Boncz. Database architectures evolution: Mammals flourished long before dinosaurs became extinct. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1648–1653, 2009.

[184] M. McAuliffe, M. Carey, and M. Solomon. Towards effective and efficient free space management. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 398–400, 1996.

[185] P. McKenney. Memory barries: A hardware view for software hackers. freely available, 2010.

[186] N. Megiddo and D. Modha. Outperforming LRU with an adpative replacement cache algorithm. *IEEE Computer*, 37(4):58–65, 2004.

[187] S. Mehta. *Scalable Compiler Optimizations for Improving Memory System Performance in Multi- and Many-core Processors*. PhD thesis, U Minnesota, 2014.

[188] P. Menon, T. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *VLDB Journal*, 11(1):1–13, 2017.

[189] U. Meyer, P. Sanders, and J. Sibeyn. *Algorithms for Memory Hierarchies*. LNCS 2625. Springer, 2003.

[190] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *ACM Symp. on Par. Alg. and Arch. (SPAA)*, pages 73–82, 2002.

[191] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[192] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *SIGIR*, pages 274–285, 1992.

[193] I. Müeller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1123–1136, 2015.

[194] W. Mula, N. Kurz, and D. Lemire. Faster population counts using AVX2 instructions. *arXiv*, 1611.07612v3, 2016.

[195] S. Müller and H. Plattner. An in-depth analysis of data aggregation cost factors in a columnar in-memory database. In *DOLAP*, pages 65–72, 2012.

[196] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 468–478, 1988.

[197] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[198] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Engineering Bulletin*, 37(1):3–11, 2014.

[199] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 387–396, 1991.

[200] W. Ng and C. Ravishankar. Relational database compression using augmented vector quantization. In *Proc. IEEE Conference on Data Engineering*, pages 540–549, 1995.

[201] C. Nyberg, T. Barclay, Z. Cventanovic, J. Gray, and D. Lomet. AlphaSort: A RISC machine sort. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 233–242, 1994.

[202] E. O'Neil, P. O'Neil, and G. Weikum. An optimality proof for the LRU-k page replacement algorithm. *Journal of the ACM*, 46(1):92–112, 1999.

[203] O.Polychroniou and K. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2014.

[204] O.Polychroniou and K. Ross. Vectorized Bloom filters for advanced SIMD processors. In *Int. Workshop on Data Management on New Hardware (DaMoN)*, 2014.

[205] O.Polychroniou and K. Ross. Efficient lightweight compression alongside fast SIMD. In *Int. Workshop on Data Management on New Hardware (DaMoN)*, 2015.

[206] R. Pagh and F. Rodler. Cuckoo hashing. *J. of Algorithms*, 51:122–144, 2004.

[207] S. Pantela and S. Idreos. One loop does not fit all. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 2073–2074, 2015.

[208] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krueger. How to juggle columns: An entropy-based approach for table compression. In *IDEAS*, 2010.

[209] J. Patel, M. Carey, and M. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 56–66, 1994.

[210] M. Patrascu and M. Thorup. The power of tabulation hashing. *J. ACM*, 59(3):14, 2012.

[211] D. Patterson and J. Hennessy. *Computer Organization and Design: ARM Edition*. Morgan Kaufmann, 2017.

[212] P. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33:677–680, 1990.

[213] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. CPU and Cache efficient management of memory-resident databases. In *Proc. IEEE Conference on Data Engineering*, pages 14–25, 2013.

[214] O. Polychroniou, A. Raghavan, and K. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1493–1508, 2015.

[215] L. Pyeatt. *Modern Assembly Language Programming with the ARM Processor*. Newnew, 2016.

[216] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman. Main-memory scan sharing for multi-core CPUs. *Proc. of the VLDB Endowment (PVLDB)*, pages 610–621, 2008.

[217] M. V. Ramakrishna and J. Zobel. Performance in practice of string hashing functions. In *DASFAA*, pages 215–224, 1997.

[218] V. Raman, G. Swart, L. Quian, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proc. IEEE Conference on Data Engineering*, pages 60–69, 2008.

[219] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *Proc. IEEE Conference on Data Engineering*, 2006.

[220] J. Rao and K. Ross. Making b^+-trees cache conscious in main memory. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 475–486, 2000.

[221] L. Rashid, W. Hassanein, and M. Hammad. Exploiting multithreaded architectures to improve the hash join operation. In *MEDEA*, pages 46–53, 2008.

[222] G. Ray, J. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[223] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. of the VLDB Endowment (PVLDB)*, 9(3), 2015.

[224] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 47–57, 2001.

[225] K. Ross. Conjunctive selection conditions in main memory. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 108–120, 2002.

[226] K. Ross. Selection conditions in main memory. *ACM Trans. on Database Systems*, 23(1):132–161, 2004.

[227] K. Ross. Efficient hash probes on modern processors. In *Proc. IEEE Conference on Data Engineering*, pages 1297–1301, 2007.

[228] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Int. Symp. on Computer Architecture*, pages 111–121, 1999.

[229] M. Roth and S. Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[230] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake joins. *Proc. of the VLDB Endowment (PVLDB)*, 7(9):709–720, 2014.

[231] G. Sacco. Index access with a finite buffer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 301–309, 1887.

[232] G. Sacco and M. Schkolnick. A technique for managing the buffer pool in a relational system using the hot set model. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–262, 1982.

[233] G. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, 1986.

[234] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, pages 34–40, 2010.

[235] F. Schuhknecht, P. Khanchandani, and J. Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *PVLDB*, 8(9), 2015.

[236] L. Shapiro, S. Ni, and G. Graefe. Full-time data compression: An adt for database performance. Unpublished, 1993.

[237] A. Shatdal, C. Kant, and J. Naughton. Cache conscious algorithms for relational query processing. Technical Report 1234, U Wisconsin, Madison, 1994.

[238] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. of Exp. Algorithmics*, 9(1.5), 2004.

[239] B. Sinharoy, J. Van Norstrand, R. Eickemeyer, H. Le, J. Leenstra, D. Nguyen, B. Königsburg, K. Ward, M. Brown, J. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. Bishop, M. Gschwind, M. Boersma, M. Kaltenbach, T. Karkhanis, and K. Fernsler. IBM POWER8 processor core microarchitecture. *IBM J. Res. & Dev.*, 59(1):2, 2015.

[240] J. Sompolski. *Just-in-time Compilation in Vectorized Query Execution.* PhD thesis, University of Warsaw, 2011.

[241] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Int. Workshop on Data Management on New Hardware (DaMoN)*, 2011.

[242] D. Sorin, M. Hill, and D. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, 2011.

[243] W. Starke, J. Stuecheli, D. Daly, J. Dodwon, F. Auernhammer, P. Sagmeister, G. Guthrie, C. Marino, M. Siegel, and B. Blaner. The cache and memory subsystem of the IBM POWER8 processor. *IBM J. Res. & Dev.*, 59(1):1, 2015.

[244] A. Stepanov, A. Gangolli, D. Rose, R. Ernst, and P. Oberoi. SIMD-based decoding of posting lists. In *CIKM*, page 317, 2011.

[245] B. Stroustrup, editor. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[246] M. Switakowski, P. Boncz, and M. Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12):1759–1770, 2012.

[247] M. Thorup. Even strongly universal hashing is pretty fast. In *SODA*, pages 496–497, 2000.

[248] M. Thorup. String hashing for linear probing. In *SODA*, pages 655–664, 2009.

[249] M. Thorup and Y. Zhang. Tabulation-based 4-universal hashing with applications to second moment estimation. In *SODA*, pages 615–624, 2004.

[250] M. Thorup and Y. Zhang. Tabulation-based 5-universal hashing and linear probing. In *ALENEX*, pages 62–76, 2010.

[251] M. Thorup and Y. Zhang. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comp.*, 41(2):293–331, 2012.

[252] S. Viglas. Just-in-time compilation for SQL query processing. In *Proc. IEEE Conference on Data Engineering*, pages 1298–1301, 2014.

[253] A. Viola. *Analysis of Hashing Algorithms and a New Mathematical Transform*. PhD thesis, 1995.

[254] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 993–1008, 2017.

[255] J. Wassenberg and P. Sanders. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.

[256] M. Wegman and J. Carter. New classes and applicatoins of hash functions. In *Ann. Symp. on Foundations of Computer Science*, 1979.

[257] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[258] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[259] T. Willhalm, I. Oukid, I. Müller, and F. Färber. Vectorizing database column scans with complex predicates. In *ADMS*, pages 1–12, 2013.

[260] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scans: Ultra fast in-memory table scan using on-chip vector processing units. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 385–394, 2009.

[261] A. Williams. *C++ Concurrency In Action*. Manning, 2012.

[262] P. Wong, Z. Feng, W. Xu, E. Lo, and B. Kao. TLB misses – the missing issue of adaptive radix tree? In *DaMoN*, 2015.

[263] S. Wu, Y. Shuf, H. Min, H. Franke, B. Iyer, F. Villafuerte, and J. Watts. Analyzing and improving table space allocation. In *Australasian Database Conference (ADC)*, 2011.

[264] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. Technical Report TR2004-1970, Cornell University, 2004.

[265] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *ACM SIGMETRICS*, pages 181–192, 2005.

[266] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *Int. Conf. on Quantitative Evaluation of Systems*, 2005.

[267] H. Zhang, G. Chen, B. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Trans. on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[268] H. Zhang, G. Chen, B. Ooi, W.-F. Wong, S. Wu, and Y. Xia. Anti-caching-based elastic memory management for big data. In *Proc. IEEE Conference on Data Engineering*, 2015.

[269] X. Zhao, R. Johnson, and N. Martin. DBJ – a dynamic balancing hash join algorithm in multiprocessor database systems. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 301–308, 1994.

[270] X. Zhao, R. Johnson, and N. Martin. DBJ – a dynamic balancing hash join algorithm in multiprocessor database systems. *Information Systems*, 19:89–100, 1994.

[271] J. Zhou and K. Ross. Implementing database operations using SIMD instructions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2002.

[272] J. Zhou and K. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 191–202, 2004.

[273] M. Zukowski, S. Heman, and P. Boncz. Architecture-conscious hashing. In *DaMoN*, 2006.