Main Memory Database Management Systems

Guido Moerkotte

Outline

1. Hardware
2. Operating System
3. Hashing
4. Compression
5. Storage Layout
6. Physical Algebra: Processing Modes
7. Expression Evaluation
8. Physical Algebra: Implementation
9. Index Structures
10. Parallelism
11. Boolean Expressions
12. Transaction Management

Introduction

The holy grail for a DBMS is one that is:

- 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.

Introduction

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

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

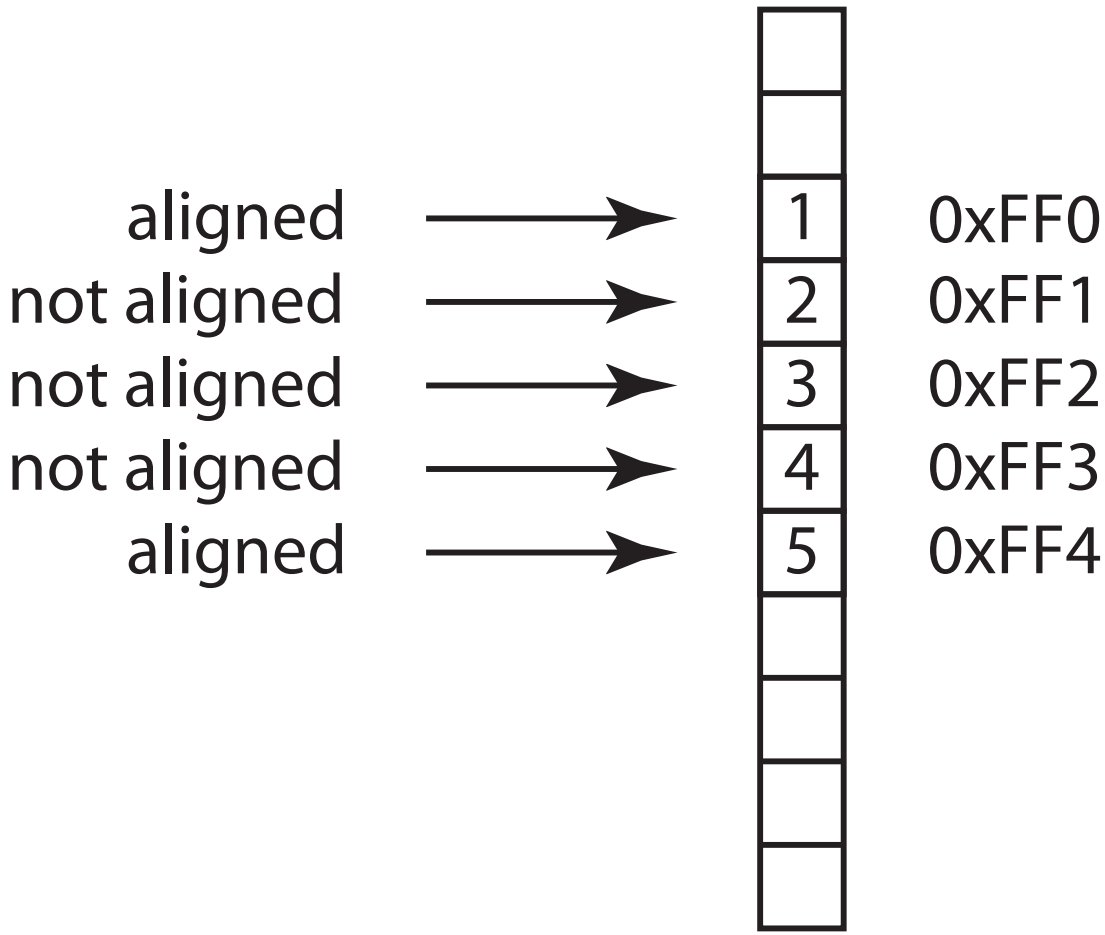
Hardware

Hardware: Alignment

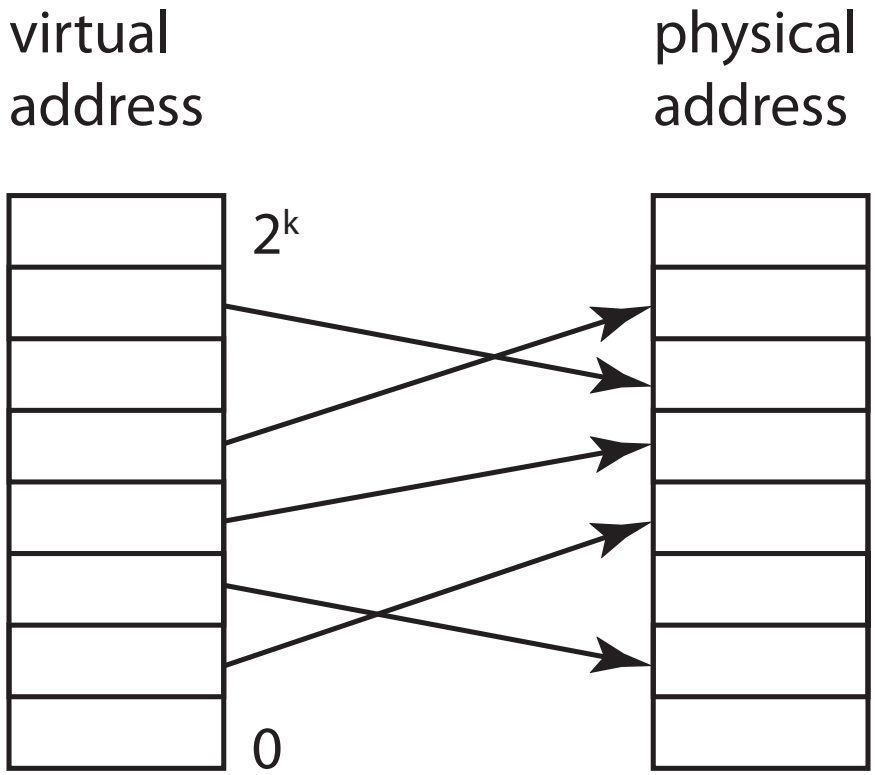
Accessing a data item d at memory address a is aligned if

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

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



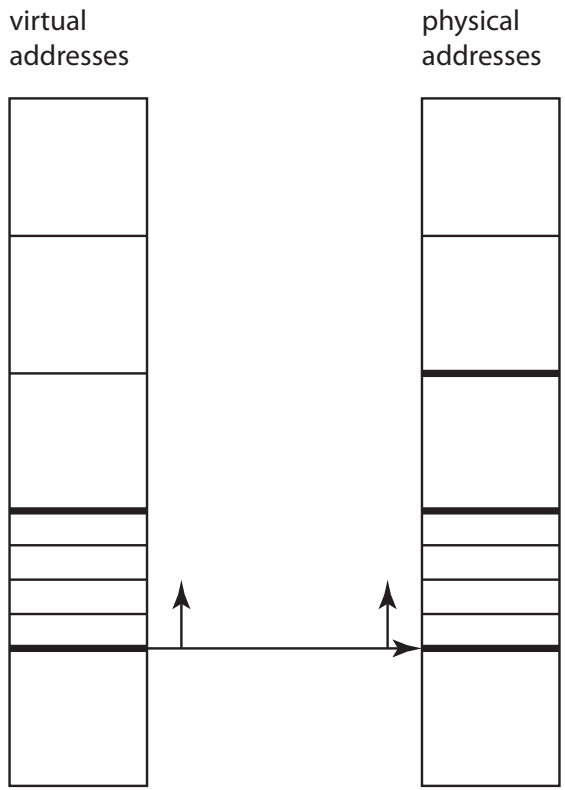
Virtual Memory



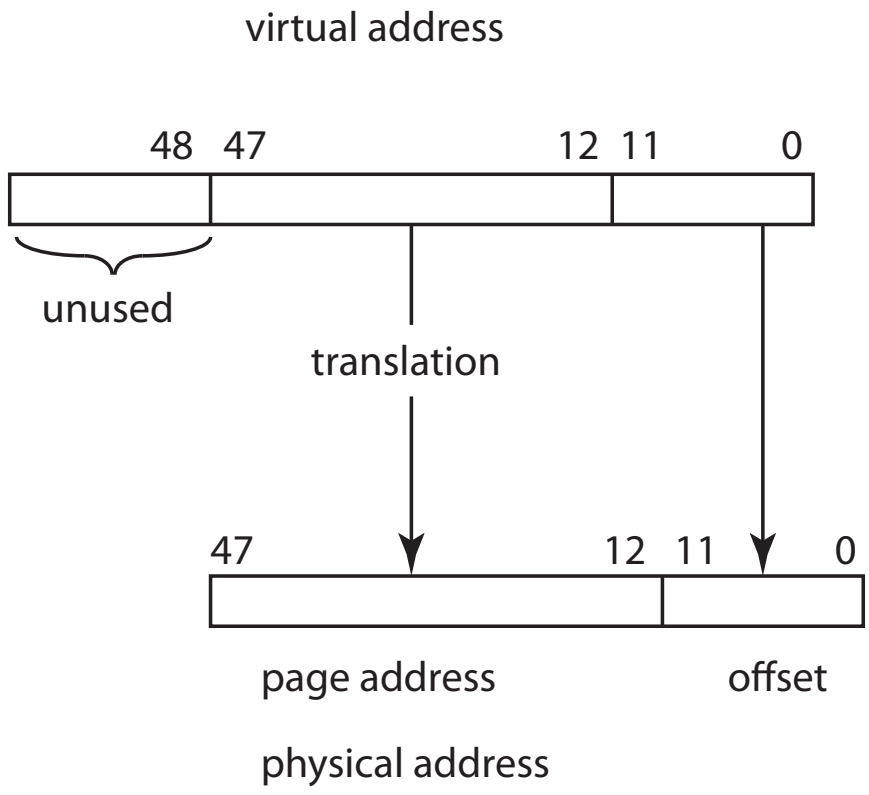
Per byte translation table is too expensive.

Virtual Memory

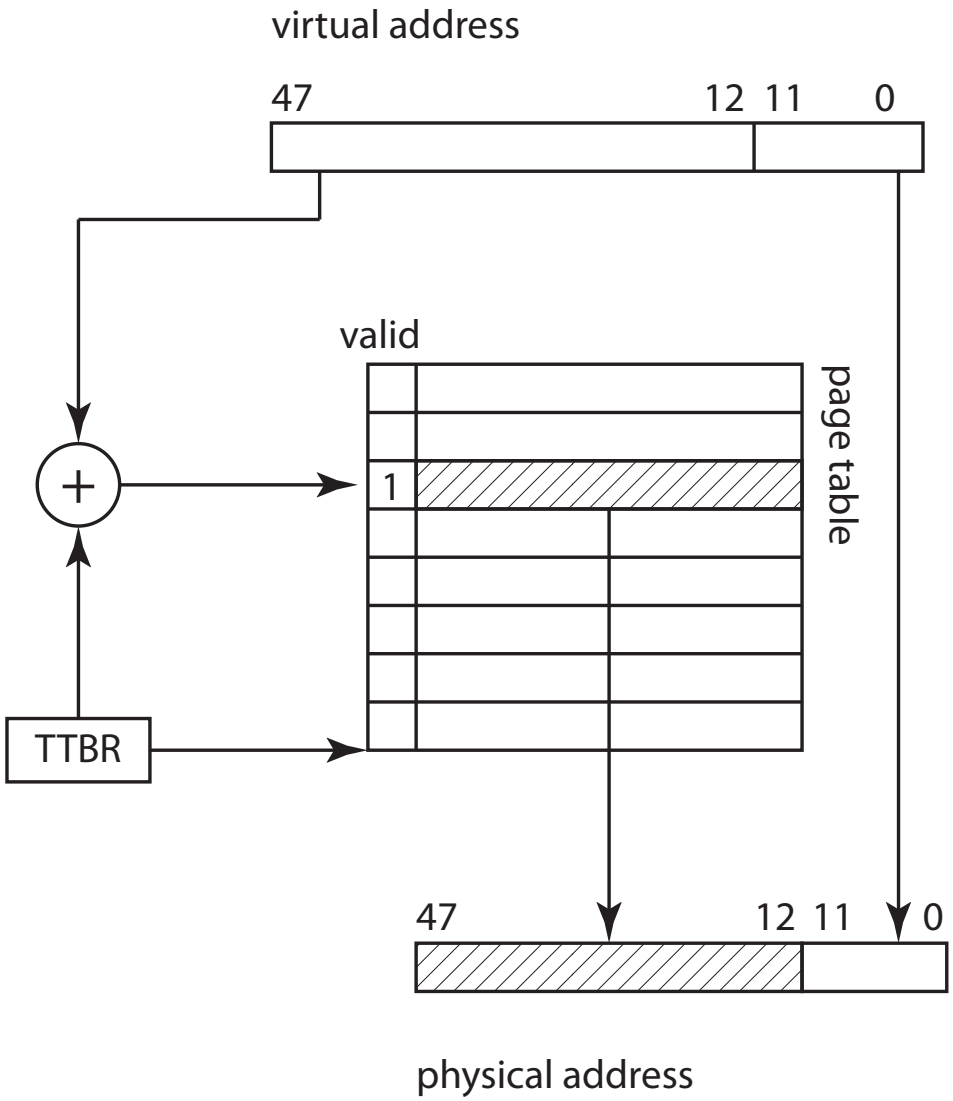
Map pages:



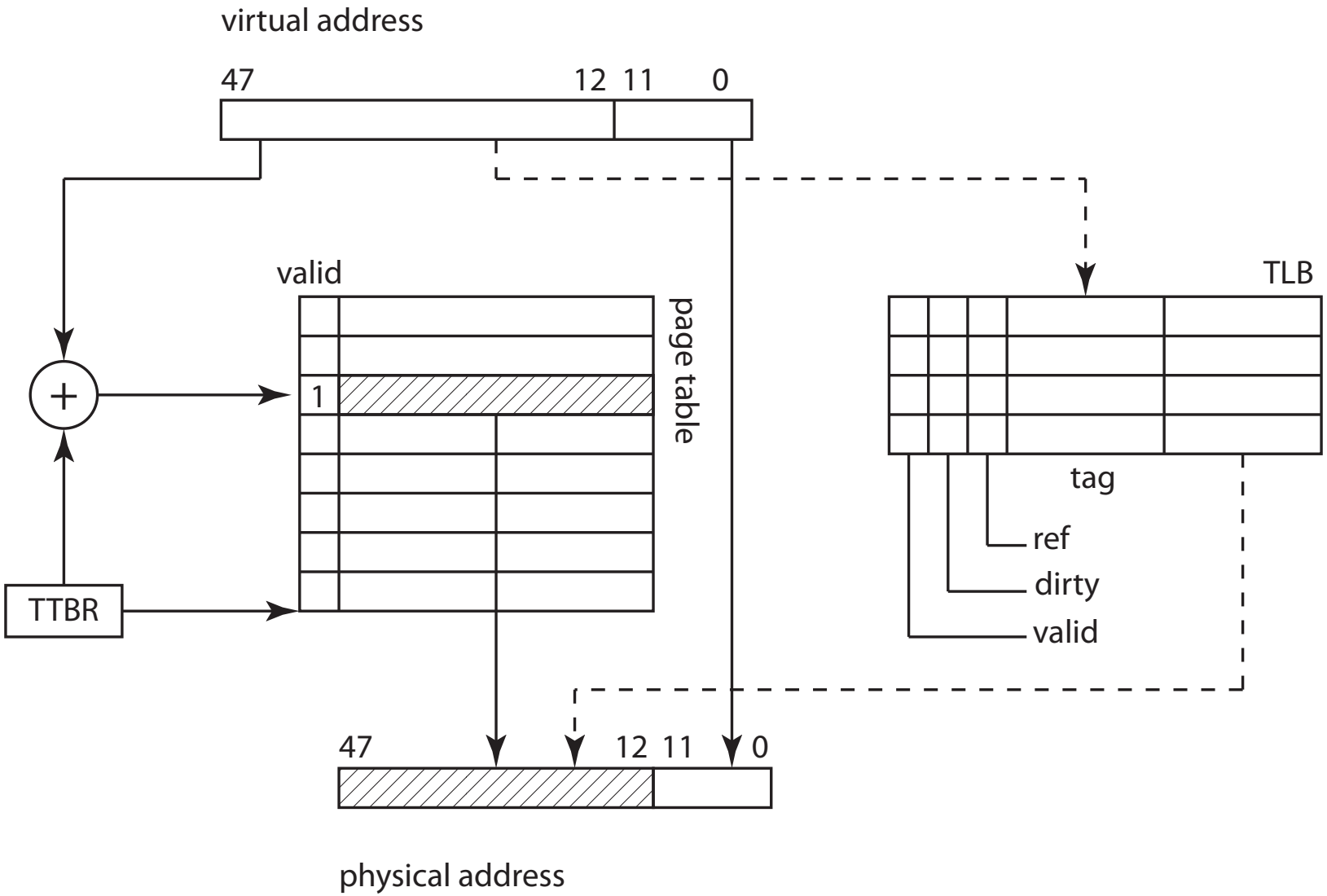
Virtual Memory: address translation



Virtual Memory: translation table base register



Virtual Memory: TLB



Virtual Memory: TLB: numbers

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

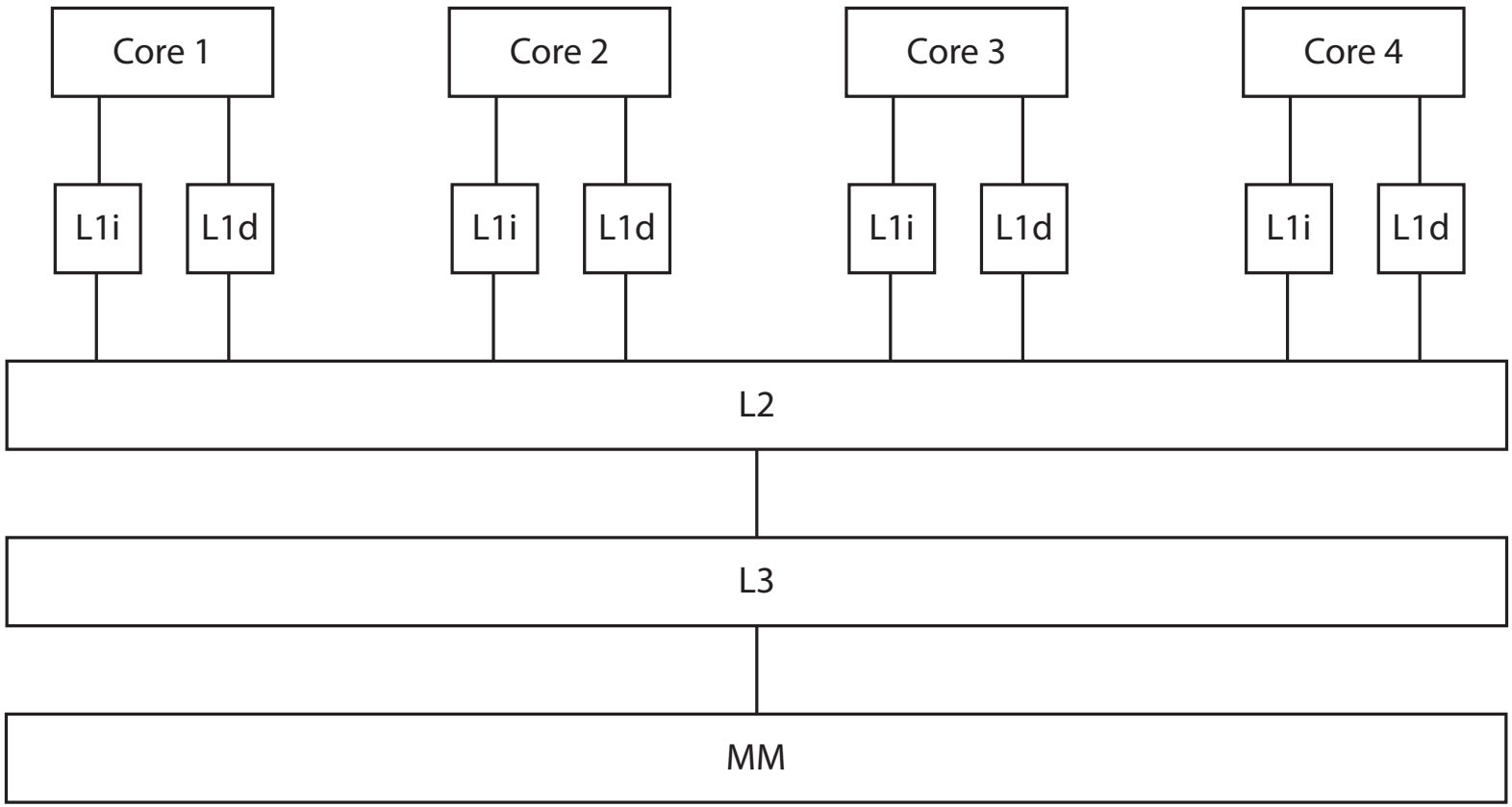
Caches: Quantitative Features

- size
- associativity
- hierarchy
- latency

Caches: 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

Caches: Sample Organization of the Memory Hierarchie

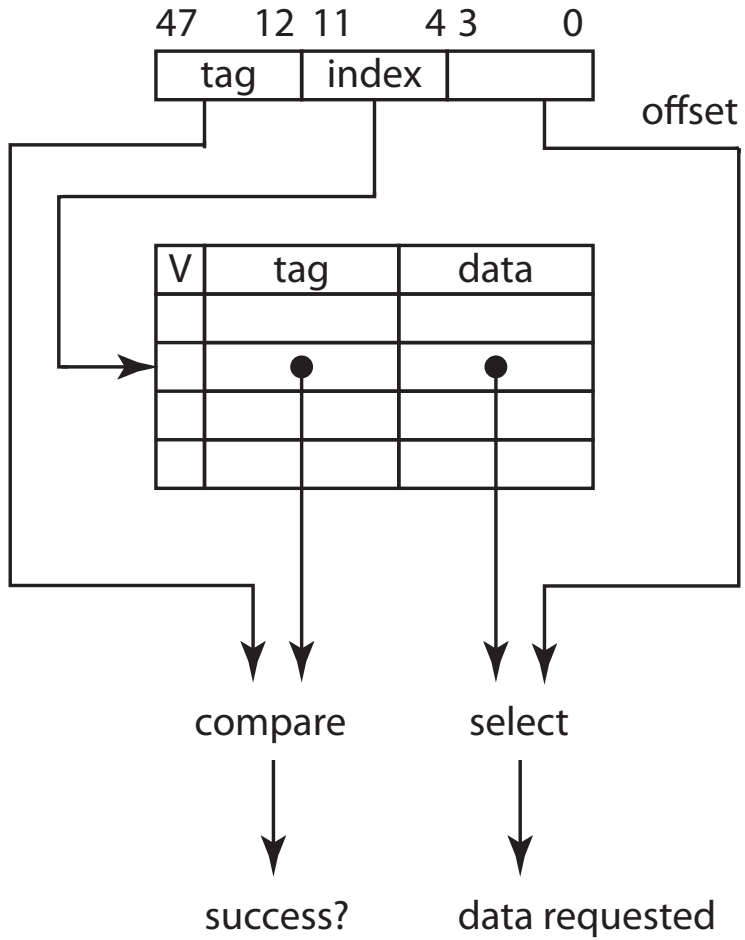


Caches: Latencies

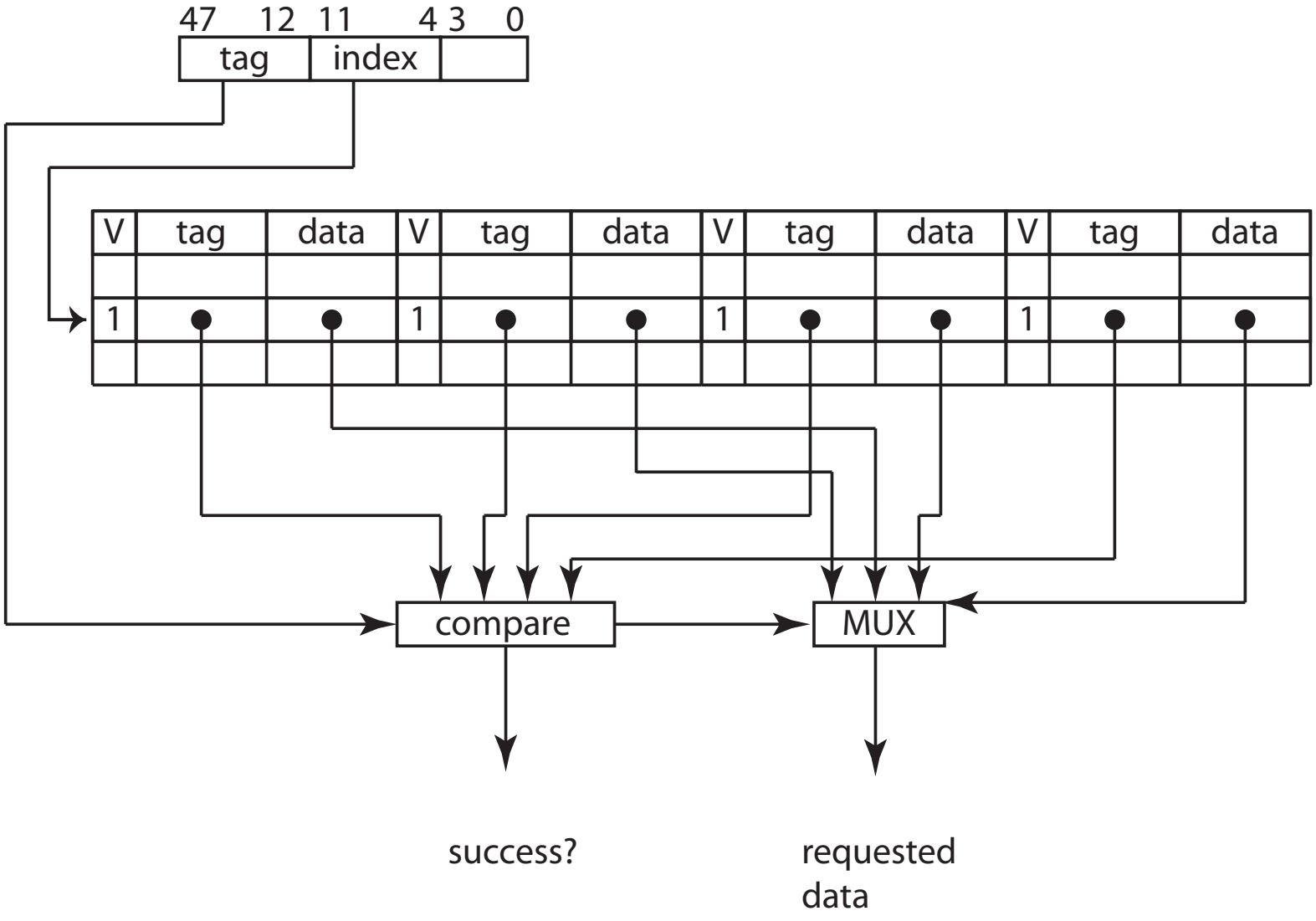
mem	latency [cycles]
register	≤ 1
L1	3-4
L2	≈ 14
TLB1	≈ 12
TLB2	≈ 30
main memory	≈ 240

(these are rough approximate numbers)

Caches: non-associative



Caches: 4-way associative



Caches: some numbers

CPU	L1i	L1d	L2	L3	L4	L1 TLB entries
	KB	KB	KB	MB	MB	
Power8	32	64	512	8*	16**	72i + 48d
Xeon E5 v4	32	32	256	2.5*		128i + 64d
i7-4790	32	32	256	8*		64i + 64d
Exynos 2254	32	32	2048*	-		32i + 32d

*: shared; **: per buffer chip

Prefetching: Hardware

Hardware prefetcher:

- adjacent cache line prefetcher
- stride prefetcher

Often: prefetchers do not prefetch across page boundaries.

Prefetching: Software

Software prefetching:

- explicit prefetch instructions

Prefetcher: Performance

Measure code fragment:

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

l index array filled in two different ways:

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

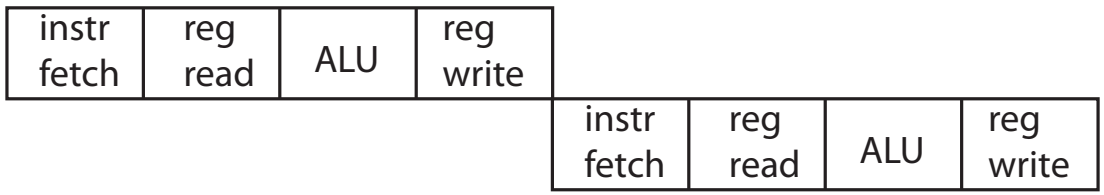
Prefetcher: Performance

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

CPU: Pipelining

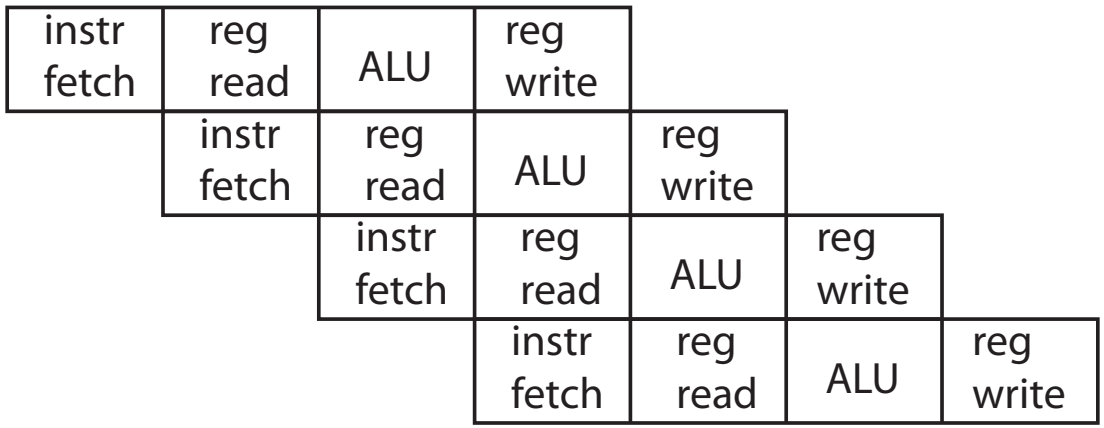
Illustration of non-pipelined execution (simplified):



We get an IPC of 0.25. (CPI of 4.)

CPU: Pipelining

Illustration of pipelined execution:



We get an IPC/CPI of 1.

CPU: Pipelining

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.

CPU: 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)

Examples:

- Haswell has 8 ports per core
- Power8 has 16 execution pipelines per core

CPU: out-of-order: read

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. 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;
}
```

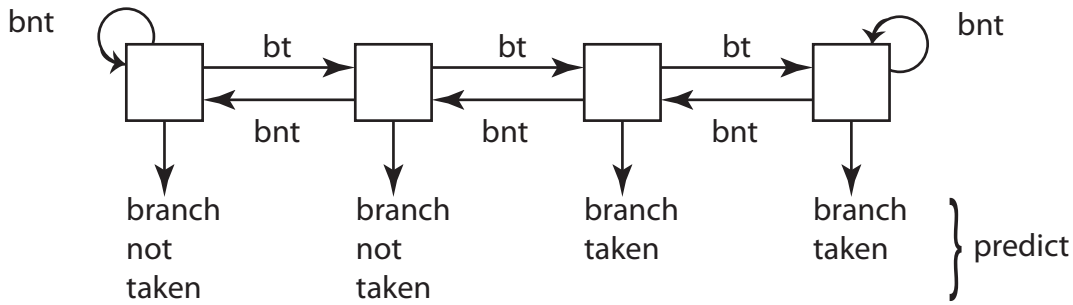
The execution times per element in the array on a i7-4790 are:

sum0	0.375 ns
sum1	0.254 ns

where we compiled with `gcc -O2`.

Branch (Mis-) Prediction

Schematic 2-bit branch predictor:



BMP: code fragment with branching

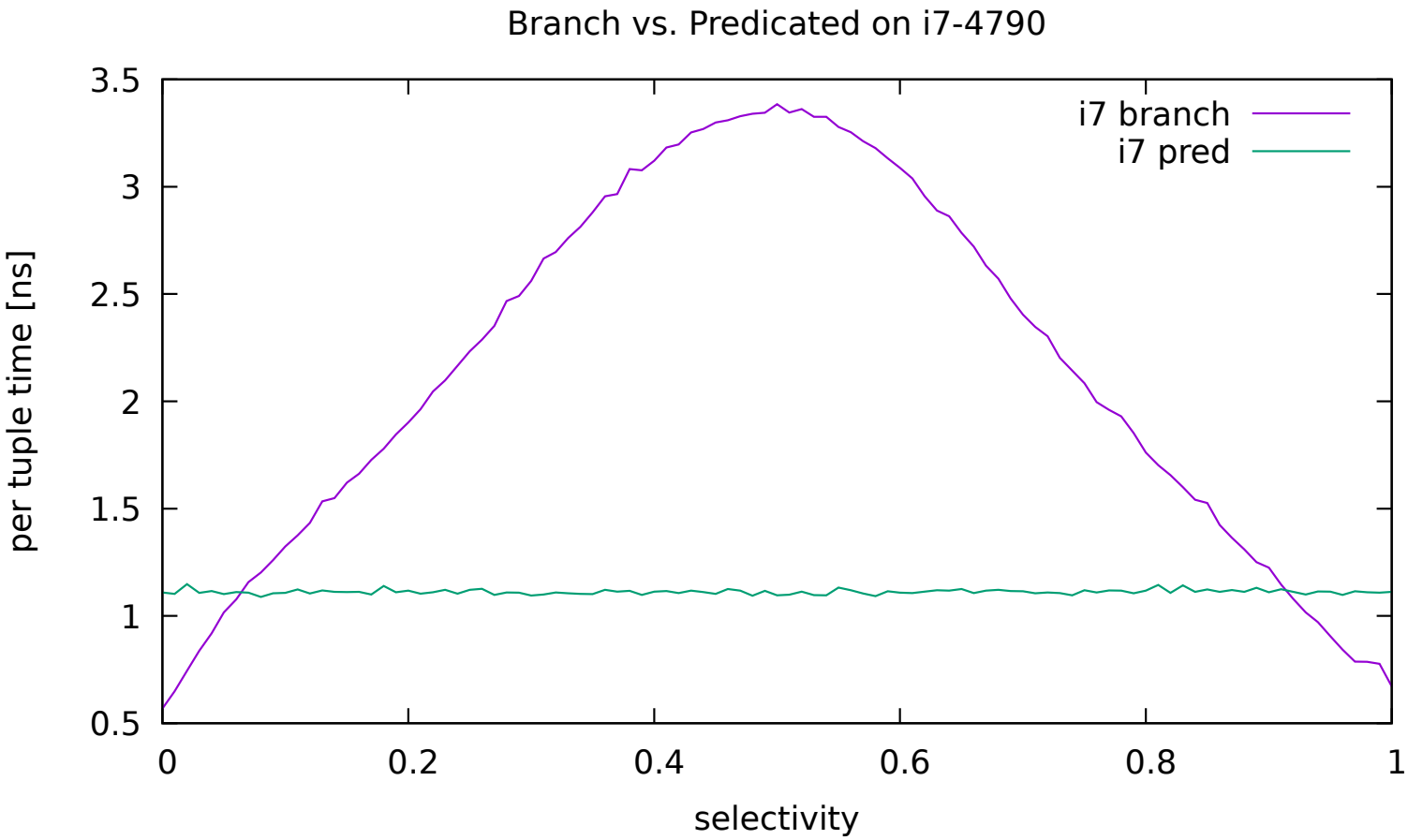
```
SELECT(int* b, int* a, int l, int n)
1   int j = 0;
2   for (int i = 0; i < n; ++i)
3         if (a[i] < l)
4               b[j++] = a[i];
5   return j
```

BMP: predicated code

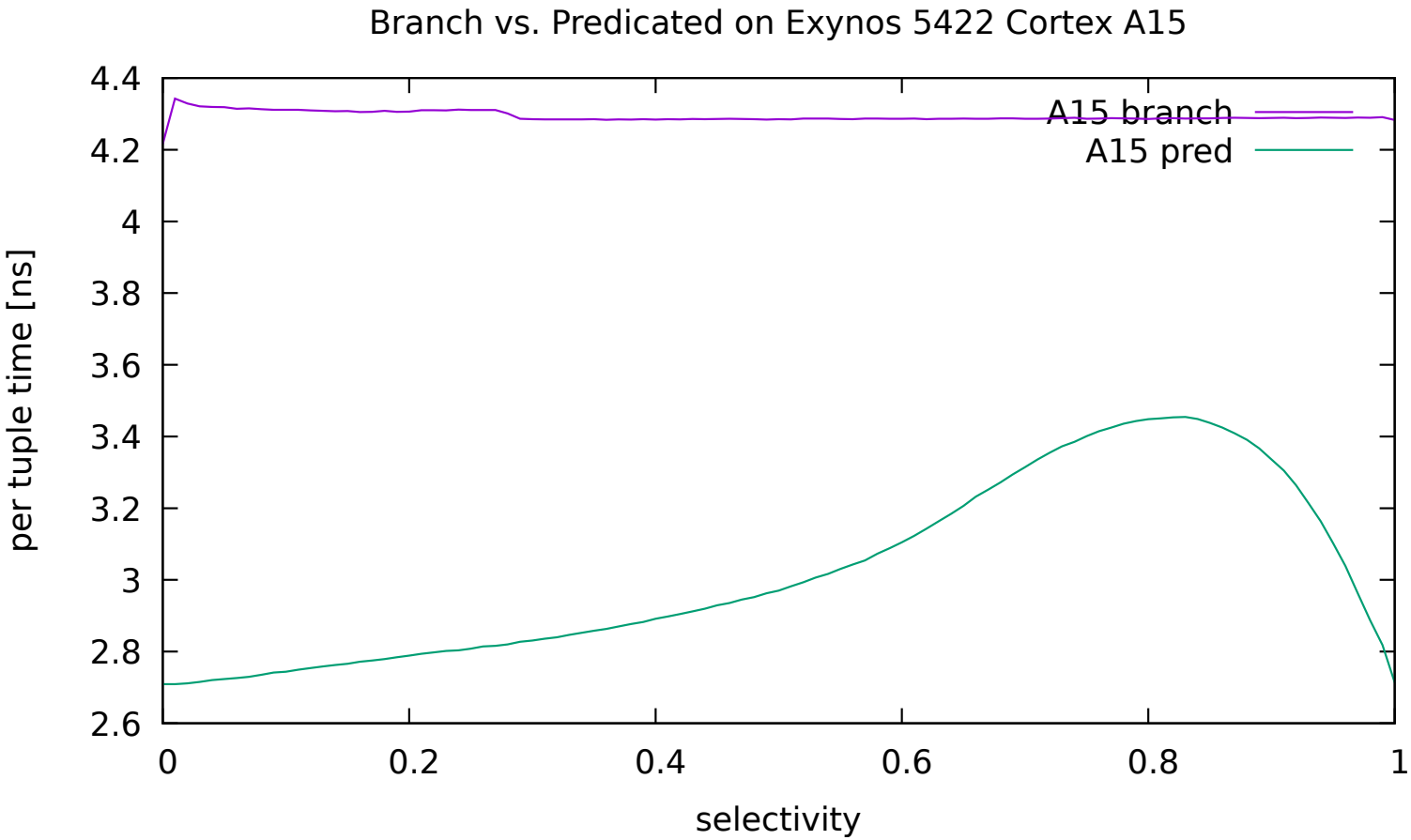
code fragment 2: predicated code suggested by Ross:

```
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
```

BMP: i7-4790


Branch vs. Predicated on i7-4790
i7 branch
i7 pred
per tuple time [ns]
selectivity
3.5
3
2.5
2
1.5
1
0.5
0
0.2
0.4
0.6
0.8
1

BMP: Samsung Exynos 6422 Cortex A15



[Note: conditional execution of instructions on ARM]

SIMD

Idea:

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

Supported by virtually all processors:

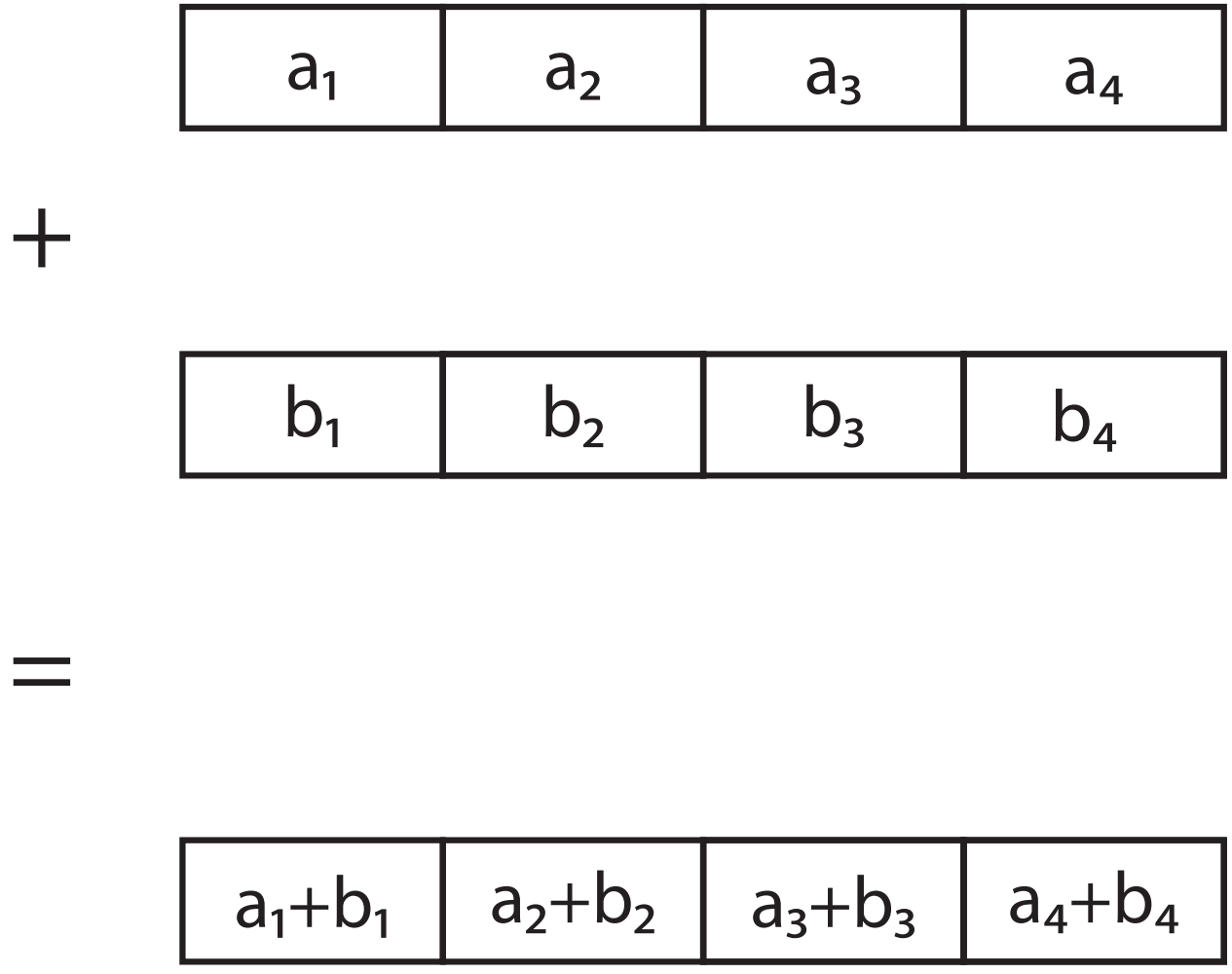
- ARM: NEON
- Intel: SSE, AVX
- Power: VMX, VSX
- Sparc: VIS

Usable

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

SIMD: Idea

Example illustration of a SIMD `add` operation:



SIMD: Intel: Intrinsics: Overview

- 128 bit, 256 bit, 512 bit SIMD registers
- arithmetics, comparisons, bit operations
- load, broadcast load, store, all selective/masked
- scatter/gather, conflict detection

SIMD: Intel: 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
```

SIMD: Intel: gather

- __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
```

SIMD: Intel: scatter

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

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
```

SIMD: Intel: scatter: conflict

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

- _mm256_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
```

SIMD: Intel: 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
```

SIMD: Intel: compare: collect result

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
```

Bit Manipulations (1)

Examples:

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

Bit Manipulation (2)

other useful instructions accessible by builtins are:

$$
\begin{array}{rcll}
\texttt{blsr}(a) & := & a \mathbin{\textcircled{\wedge}} (a-1) & \text{// reset lowest bit set} \\
\texttt{blsi}(a) & := & a \mathbin{\textcircled{\wedge}} (-a) & \text{// extract lowest bit set} \\
\texttt{blsmsk}(a) & := & a \mathbin{\textcircled{\otimes}} (a-1) & \text{// set all lower bits up to incl. lowest bi} \\
\texttt{tzcnt}(a) & & & \text{// count number of trailing zero bits} \\
\texttt{lzcnt}(a) & & & \text{// count number of leading zero bits}
\end{array}
$$

Software 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 Store

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*.

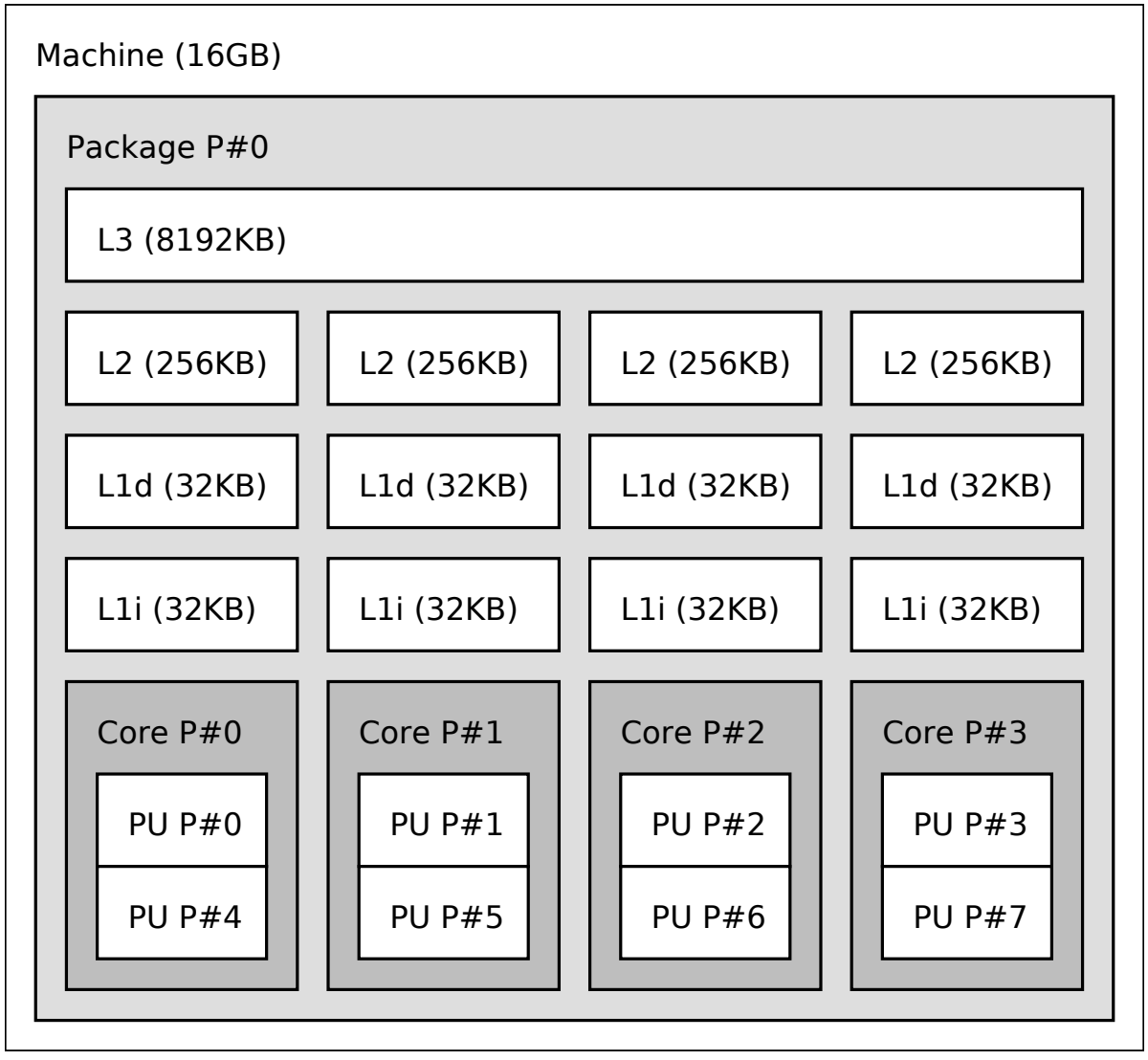
Simultaneous multithreading (SMT)

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

Notes:

- threads share the core's resources
- sometimes useful to hide latencies

Example architecture on the next slide.


Machine (16GB)
Package P#0
L3 (8192KB)
L2 (256KB)
L2 (256KB)
L2 (256KB)
L2 (256KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core P#0
PU P#0
PU P#4
Core P#1
PU P#1
PU P#5
Core P#2
PU P#2
PU P#6
Core P#3
PU P#3
PU P#7

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

Cache Coherence: MESI Protocol

The most commonly used 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

Cache Coherence: Experiment

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;
}
```

Runtime results ($n = 10^9$):

CPU	HW thread no HWT 1	HWT 2	exec time for pointer distance 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

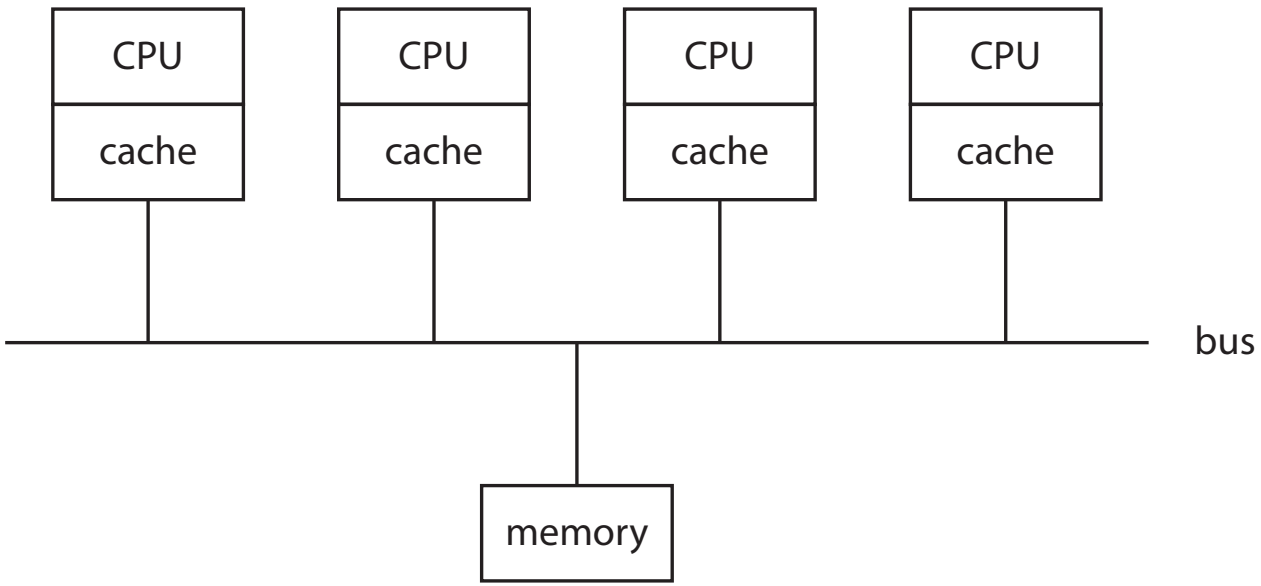
Recall: Intel i7-4790 supports SMT: hw-threads $[0, 4]$ are on core 0, $[1, 5]$ are on core 1, $[2, 6]$ on core 2, $[3, 7]$ on core 3.

Synchronization Primitives

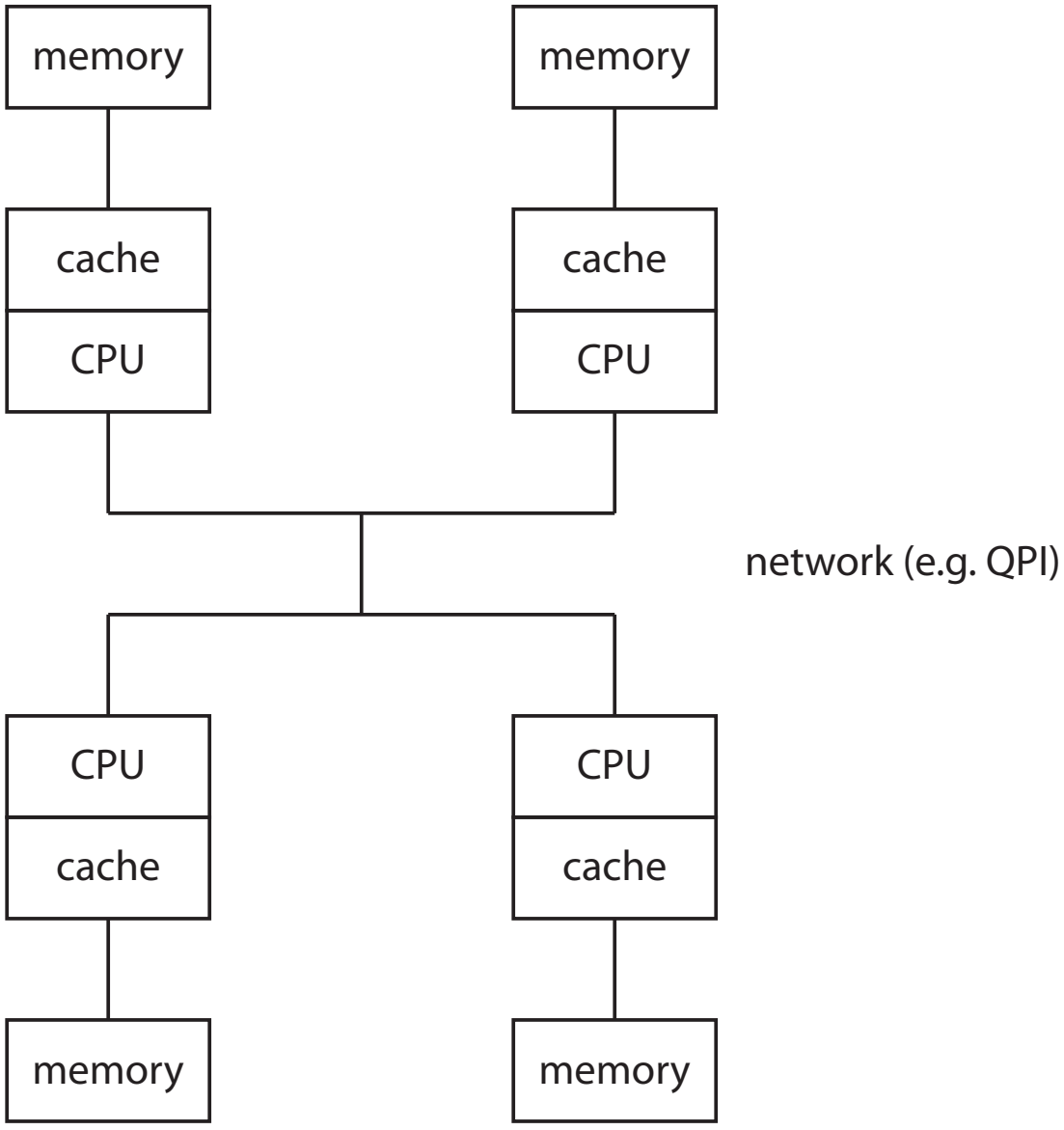
In order to synchronize different threads and prevent race conditions, synchromization primitives such as `mutex` and `semaphore` must be used and implemented. This is facilitated by atomic operations provided by the underlying hardware. Typical operations implementing atomicity are:

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

NUMA: UMA



NUMA: NUMA


memory
memory
cache
cache
CPU
CPU
network (e.g. QPI)
CPU
CPU
cache
cache
memory
memory

NUMA Measurements

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

NUMA Measurements

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

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

Performance Monitoring Unit (PMU)

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

Tools: Linux: perf

Operating Systems

useful system calls for

1. process/thread support, binding threads to cores (e.g. fork (copy on write))
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. clock access
7. hardware inspection

Operating System: Linux: hardware inspection: commands

cat /proc/cpuinfo	information about cpu/cache/ram
lscpu/lsblk/lsusb/lshw	describe hardware pieces
nproc	number of hw-threads
lstopo	topology of computer
cpuid	more information about the cpu
sensors/hddtemp	temperature and other sensors
hdparm	more information about SATA devices
dpkg-architecture	cpu, os, architecture, endianness (debian/ubuntu)
getconf	os configuration (e.g. getconf PAGESIZE)
uname	hostname, os

Operating System: Linux: hardware inspection: system calls

ioctl	everything concerning I/O
uname	as uname above
sysinfo	cpu load and memory information (total, free, swap)
sysconf	as getconf above
/proc	for everything

Hash Functions and Hash Tables

Hash Functions

- division
  $h(x) := x \mod m$
- multiplicative
  $h(x) := \lfloor m(\frac{a}{w}x \mod 1) \rfloor$
- fibonacci hashing
- polynomial over prime field
  $h(x) := \sum_{i=0}^{k-1} a_i x^i \mod p$
- multiply-(add)-shift
  $h_{a,b}(x) := (ax + b) \gg (l - l_{\text{out}})$
  or plain multiplicative:
  $h_a(x) := (ax) \gg (l - l_{\text{out}})$
- murmur hashing
- tabulation hashing
- hashpjw, CityHash, Fowler-Noll-Vo, Jenkins, SpookyHash, Zobrist, Larson

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 us:

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 $n = 2^k$: hash-table size, `#F` is the number of filled entries, `#E` is the number of empty entries, `#C` is number of entries with collisions, `llps` is the length of longest probe sequence, `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-function would distribute

Same exercise with murmur hashing:

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

Hash functions: properties

Properties wanted:

1. uniformity
2. universality
3. efficiency

Hash functions: 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.

Hash functions: 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 or Corman)

Hash functions: expected Length of the Longest Probe Sequence (llps)

Let n be the number of keys, m the hash-table size, and $\alpha = n/m$ the fill-degree, and $i^{\underline{k}} = i(i-1)\dots(i-k+1)$ 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:

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

(see Gonnet 81)

Hash functions: universal

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,

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

$$\sum_{i \in B} a_i(a_i - 1)$$

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)

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

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

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. $\square$

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

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 $r = ((x \& p) + (x >> a))$
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.

k-Universal Hash-Functions

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 $i \in 1, \ldots, |H|$

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

(c, k)-Universal Hash-Functions

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

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

Dietzfelbinger

Dietzfelbinger proposes the following 2-universal class of hash-function.
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

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

Then, $\mathcal{H}$ is (c,2)-universal with $c = \frac{5}{4}$. An efficient implementation of Dietzfelbinger's hash functions was proposed by Thorup.

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.

Tabulation Hashing

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

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

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.

Hashing string values

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)

Hashing string values

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:

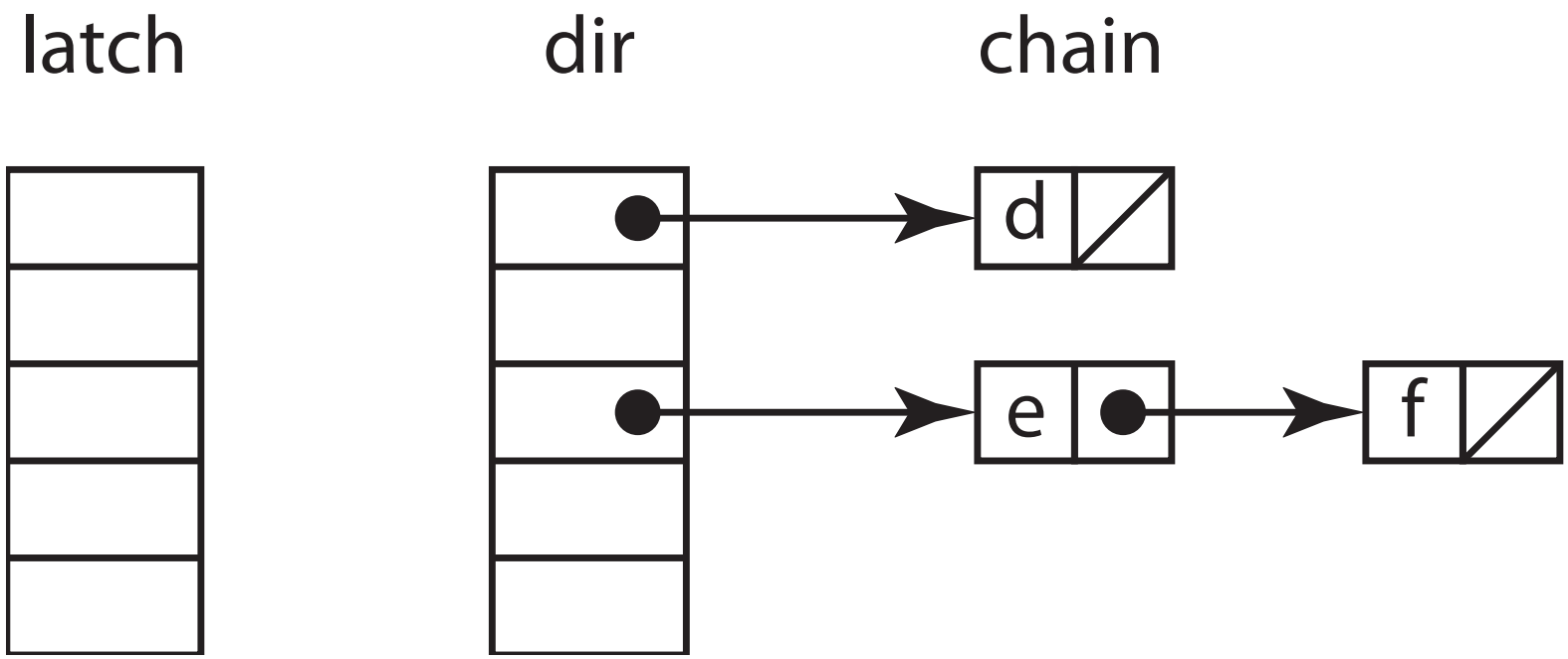
*while(*s) h = h * 101 + *s++*

Hash Table Organization

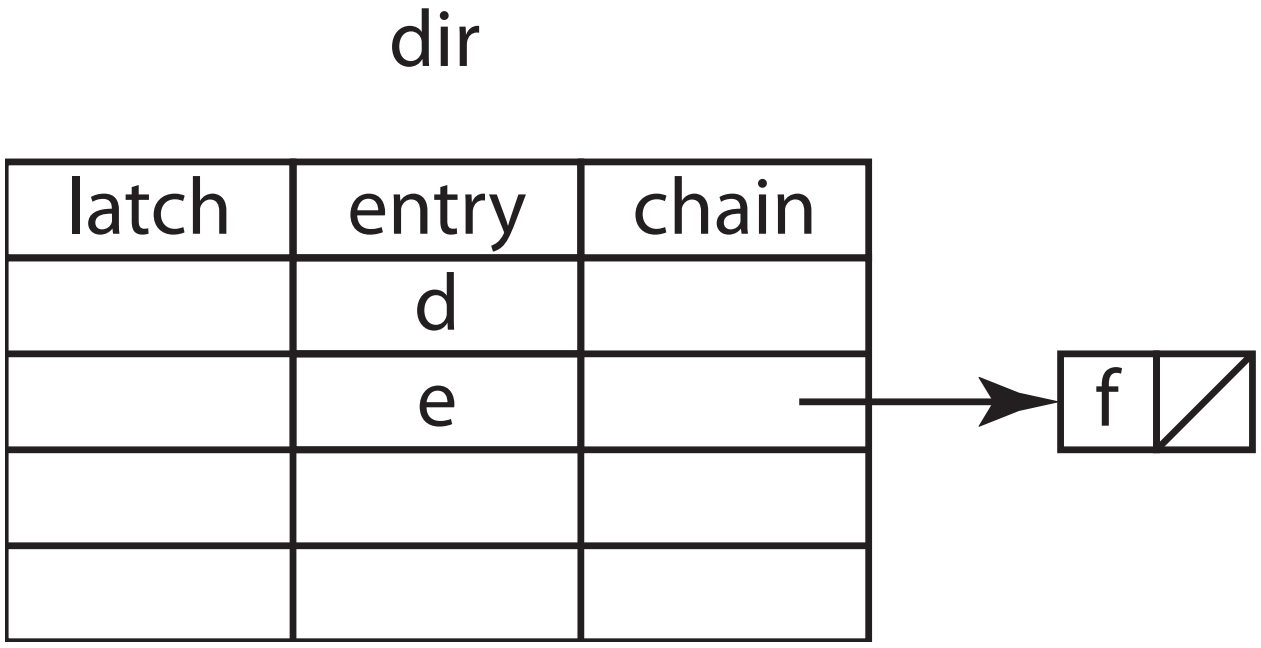
From A&D:

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

Chained Hash Table with Latches: V0

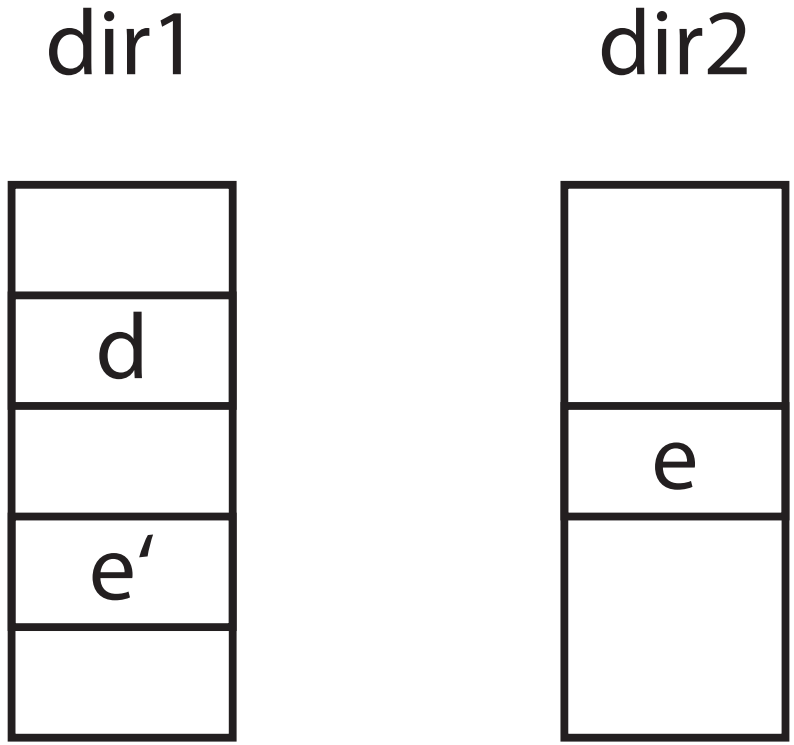


Chained Hash Table with Latches: V1



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:



Compression

light-weight compression techniques:

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

result: fixed length unsigned integers

Storage Layout

Storage Layout

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

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

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++

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.

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

Note: `rid` can be *virtual*.

Column Format (DSM) with virtual `rid`

eno
001
002
003

name
Müller
Maier
Schmidt

salary
1000
2000
4000

Column Format in C++

Column-Format in C++:

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

Query Processing: Sample Data

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).

Query Processing: Sample Query

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

Query Processing: Columns

eno
10
2000
500
700
30
8000
800

$\sigma_{100 \leq \text{eno} \leq 900}$
rid
2
3
6

$\chi_{s:salary.rid}$
300
400
700

sum
1400

Query Processing: Query in C++

```
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)
}
```

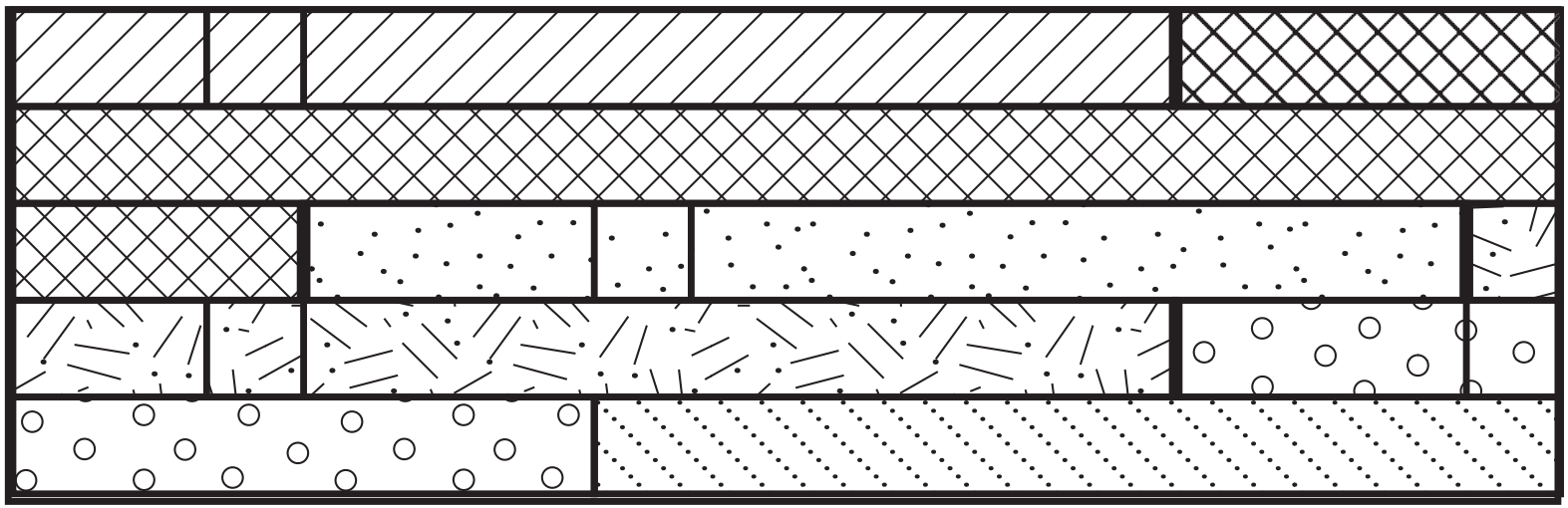
Hybrid Storage Model (PDSM)

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

Row Format and Cache Lines

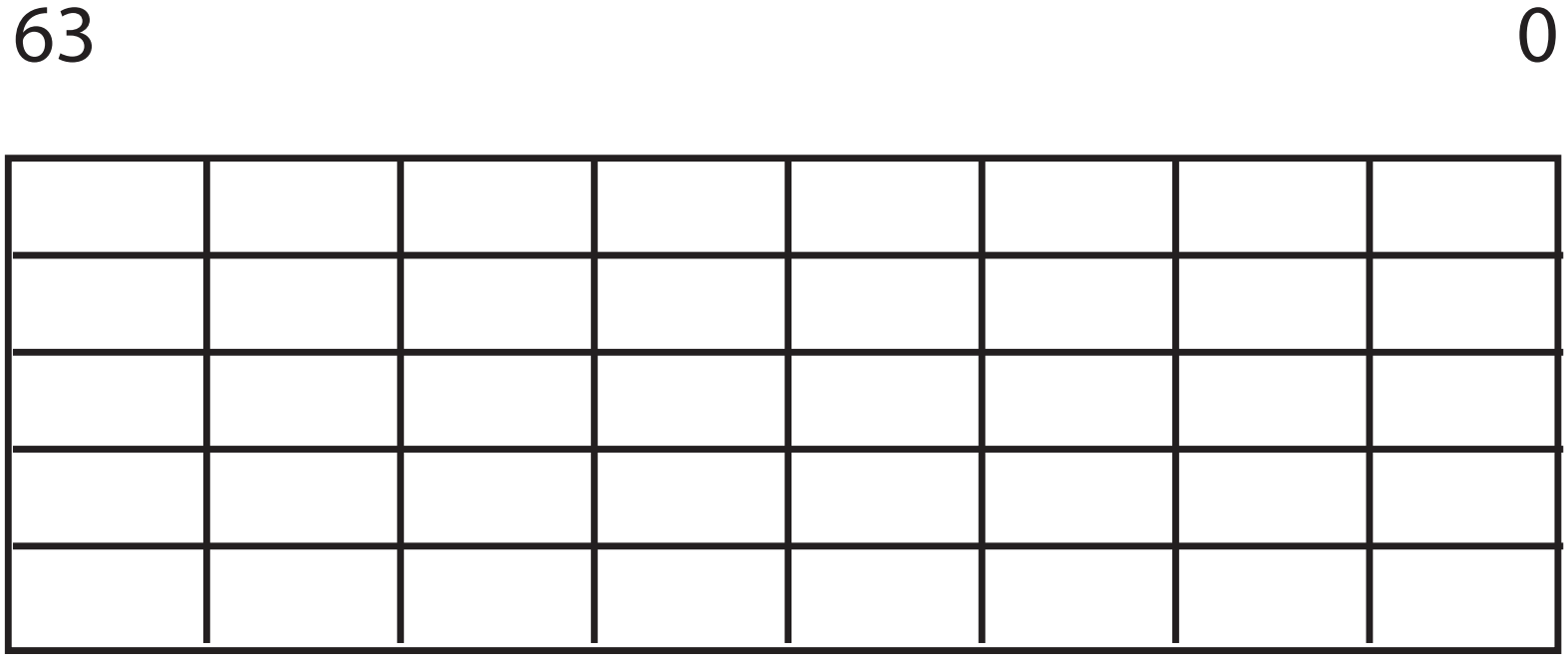
Things look bad for the row store:

63 0

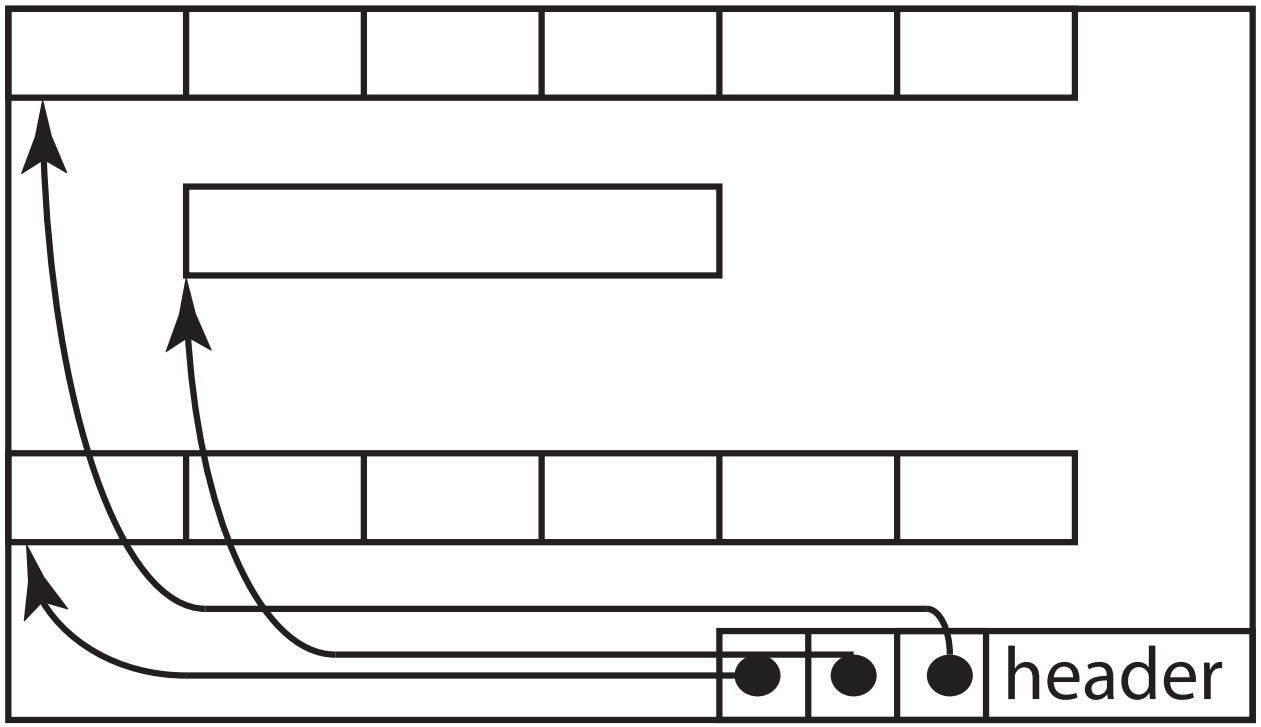


Column Format and Cache Lines

Things look good for the column store:



Putting Columns onto Pages: PAX



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.

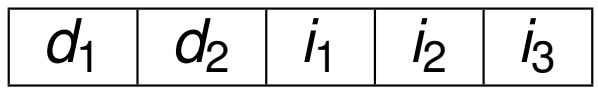
Storage Layout: Complications

fixed length: easy. Complications:

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

Row Layout: Fixed Length

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:



Row Layout: Variable Length

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

Row Layout: NULL values

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.

Row Layout: NULL Indicator (1)

Adding NULL-indicators (`nid`), everything else remains unchanged:

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.

Row Layout: NULL Indicators (2)

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...0	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...0	i_1	i_2	o_0	o_2	o_3	o_4	s_0	s_2

Row Layout: NULL Indicators (3)

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.
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.

Row Layout: Compression (1)

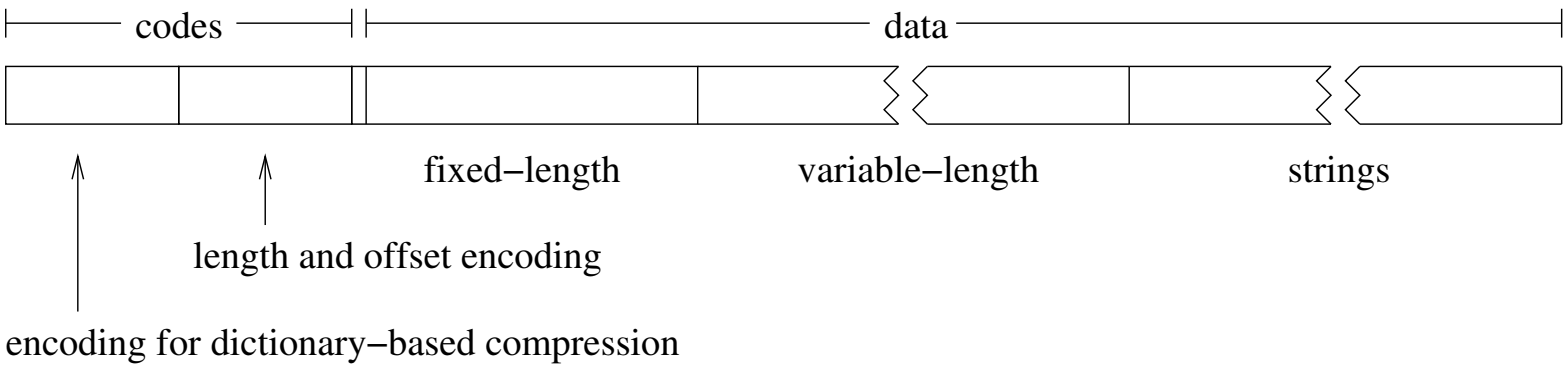
Compression adds another layer of complexity.

- 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.

We take a look at the offset-table-based approach.

Row Layout: Compression (2)

The basic record layout there is:



Row Layout: Compression (3)

For every attribute with variable length (including compressed and nullable attributes), we use a *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

Row Layout: Compression (4)

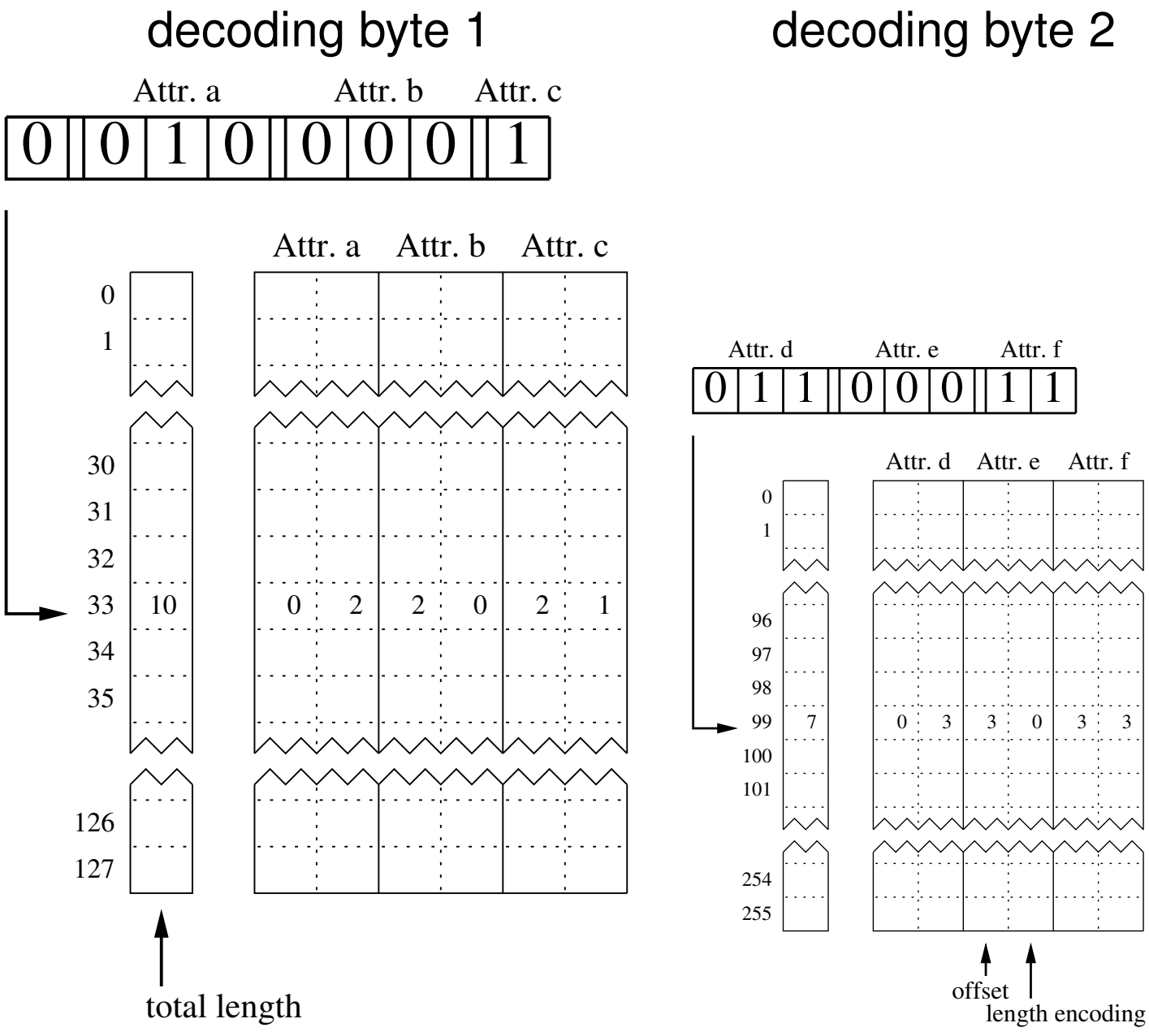
These status bits are packed together within bytes such that always all status bits belonging to a certain attributes are contained in one byte. Unused hi-bits are set to zero. Consider for example a relation with attributes

(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 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

Row Layout: Compression (5)



Row Layout: Compression (6)

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);
}
```

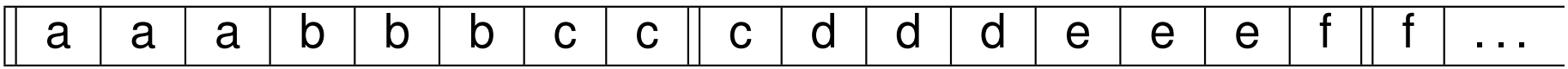
Column Layout

Many different proposals/possibilities:

- simple array
- BitPackingH
- BitSliceH
- BitSliceV
- ByteSliceV

BitpackingH

Original column layout in Hana:



Decoding:

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

BitPackingH: Decoding

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
   - 1.1 load second 128-bit register into lower part of a 256-bit register
   - 1.2 perform a 256-bit register shift
2. 4 Byte alignment:
   - 2.1 apply a shuffle operation to put four consecutive compressed values into the 4 32-bit words of a 128-bit register
3. Bit alignment:
   - 3.1 apply a shift operation with 4 individual shifts
   - 3.2 apply a bitwise AND operation with a mask to zero out irrelevant bits

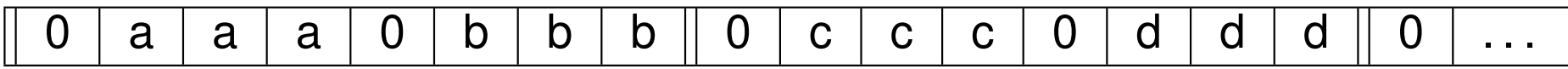
BitPackingH: Problems

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.)

BitSliceH: storage layout

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



- extra bit used to hold comparison result
- no codes spans multiple lines (padding)

Thus, if w is the line length and k the value size, both in bits, $\lfloor w/(k+1) \rfloor$ values can be stored in one register.

BitSliceH: implementing comparison operators

We discuss how comparison of a column with a value can be implemented. we use

- Let w be the (SIMD) register length.
- Let X be the register holding the column values.
- Let Y be the 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.

Further, let x and y be two k bit values.
bitwise operators: $\owedge$ bitwise and; $\owedge$ bitwise or; $\otimes$ bitwise xor; $\neg$ bitwise complement

BitSliceH: (in-)equality

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

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

Equality: complement of inequality

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

BitSliceH: less than (or equal to)

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):

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

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

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

BitSliceH: indicator bit extraction (1)

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

$$c_1 0^k \ldots c_m 0^k$$

which we wish to transform into

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

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

BitSliceH: indicator bit extraction (2)

Idea 1: successive shift/or
Example:

Input:	c_1	0	0	0	c_2	0	0	0	c_3	0	0	0	c_4	0	0	0
Step 1:	c_1	c_2	0	0	0	0	0	0	c_3	c_4	0	0	0	0	0	0
Step 2:	c_1	c_2	c_3	c_4	0	0	0	0	0	0	0	0	0	0	0	0

General procedure:

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

BitSliceH: indicator bit extraction (3)

Idea 2: replace multiple shifts by one multiplication

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

Careful: $b \leq \sqrt{w}$.

BitSliceH: converting bitvector to indices

Define two helper functions:

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

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	

BitSliceH: converting bitvector to indices (3)

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.

BitSliceH: converting bitvector to indices (3)

INPUT: BV: input bitvector, w: word width
OUPUT: L: vector of RIDs
p = 0
foreach *x* in BV
    while($x \neq 0$)
        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.

DB2 BLU

- BLINK is a row store
- DB2 BLU builds on DB2 and BLINK
- DB2 BLU can behave as a column store or a row store (PDSM)

We discuss DB2 BLU's storage model.

DB2 BLU: 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.

DB2 BLU: Overview

- 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.

DB2 BLU: Compression

- standard compression techniques
- **but** the active domain of an attribute can be partitioned
- partitioning frequency based
- compression scheme differs for each partition (e.g. number of bits)

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

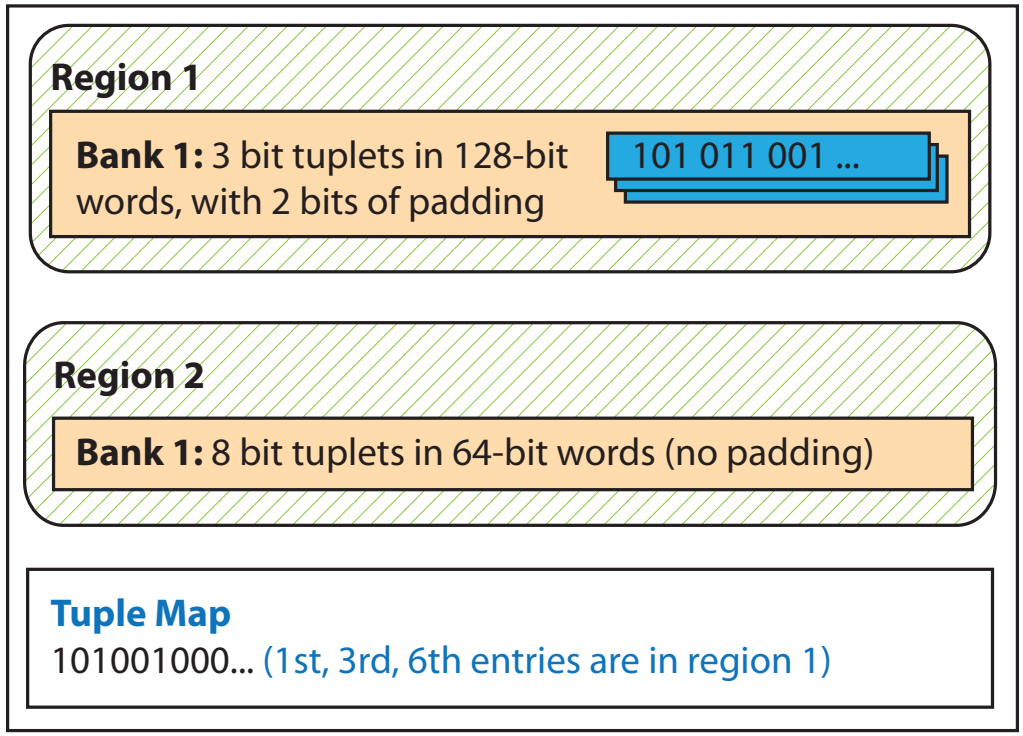
DB2 BLU: Cell/Region

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*.

DB2 BLU: Cell/Region for example



DB2 BLU: Tuple Map

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.

DB2 BLU: Banks (fixed size)

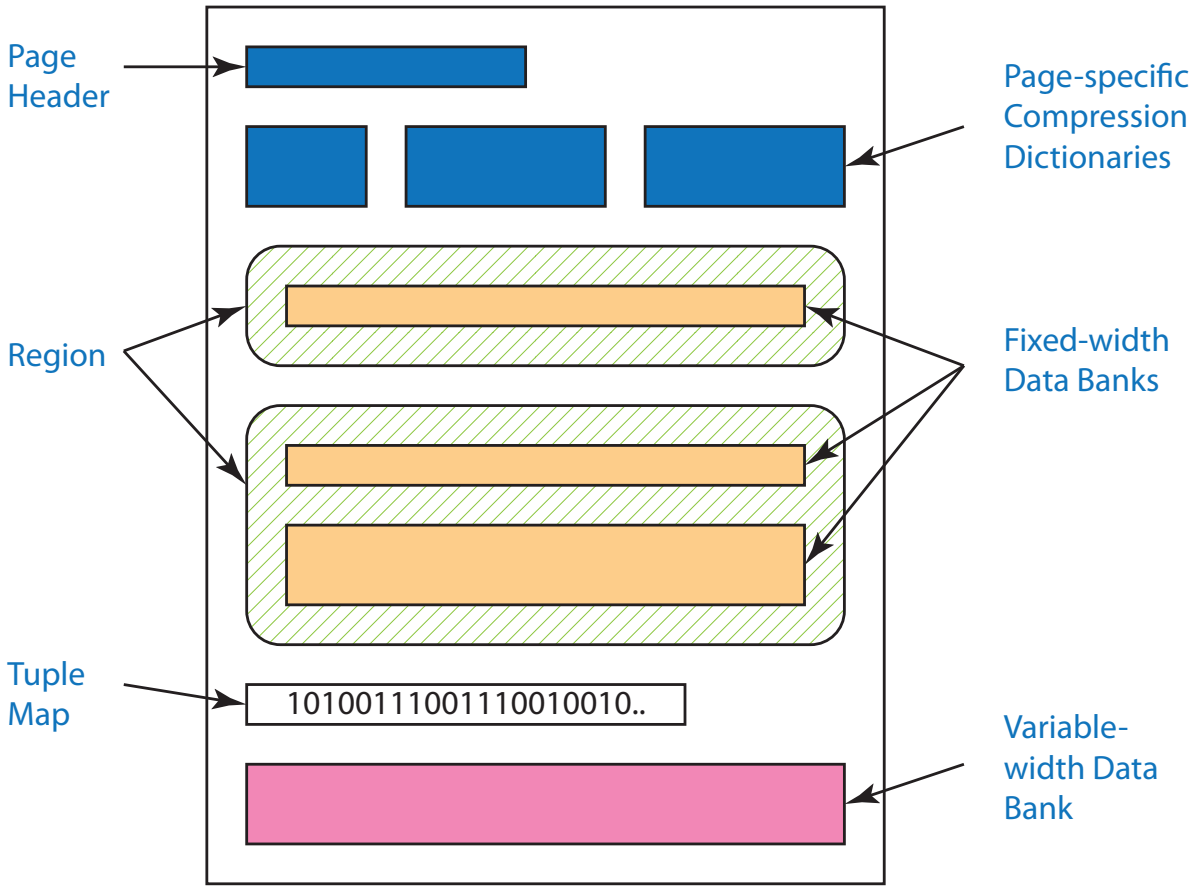
- regions are subdivided into *banks*
- banks are contiguous areas of a page (store the actual tuplets)
- tuplets do not cross bank boundaries
- bank size = 8, 16, 32, 64, 128, 256 bits

DB2 BLU: 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

DB2 BLU: Page Format



DB2 BLU: Page Level Compression

Application scenarios:

- few distinct values in some attribute
- better frame of reference

first case: page dictionaries.

DB2 BLU: Small Materialize Aggregates (SMA)

Synopses with record per page:

- page reference
- MinTSN, MaxTSN
- Min/Max column values

DB2 BLU: Table Scan

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.

SQL Server

Apollo:

- for OLAP
- originally 'column index'
- later index-only columns

Hekaton:

- for OLTP
- main-memory optimized row store

SQL Server: Apollo

- rows are divided into row groups
- each row group: segments for each column
- segments stored continously
- dictionary-based compression; bit packing or run-length encoding
- delta

SQL Server: Hekaton

Goal:

- improve OLTP throughput of SQL Server by 10x-100x

SQL Server: Hekaton: Analysis (1)

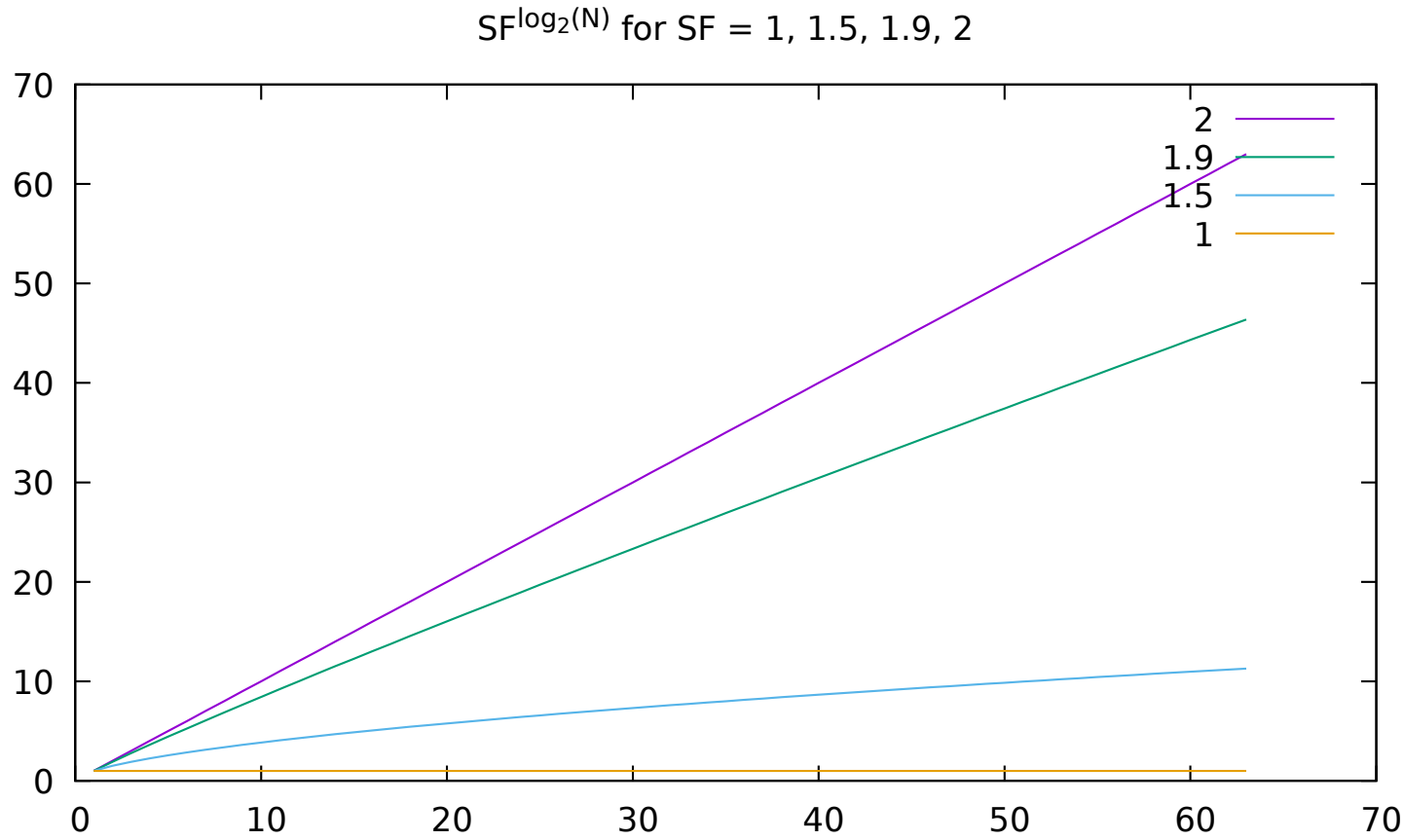
The performance of any OLTP system can be expressed as

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

where

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

Scale Factor Formel Graph



SQL Server: Hekaton: Analysis (2)

Using

IR	=	instructions retired
CPI	=	cycles per instruction

we can rewrite the above to

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

SQL Server: Hekaton: Analysis (3)

Remember:

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

Observations for SQL Server:

- 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

$$1.89^8 = 162.8$$

Ideal: factor 256. Maximum improvement:

$$256/162.8 = 1.57$$

extraordinarily good CPI 0.8 leads to factor of 2. total:

$$2 * 1.57 = 3.14$$

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

SQL Server: Hekaton: Architectural Guidelines

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

SQL Server: Hekaton: Storage Layer (1)

- Hekaton table is completely contained in main memory
- two types of indexes:
  - Bw-Tree (latch-free B-Tree)
  - hash index (latch-free hash table)
- a table can have multiple indexes
- record lookup is always by index

SQL Server: Hekaton: Storage Layer (3)

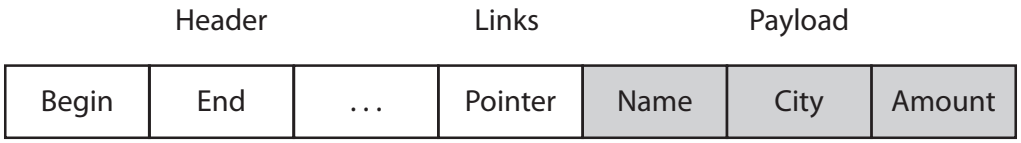
Example: Bank Account:

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

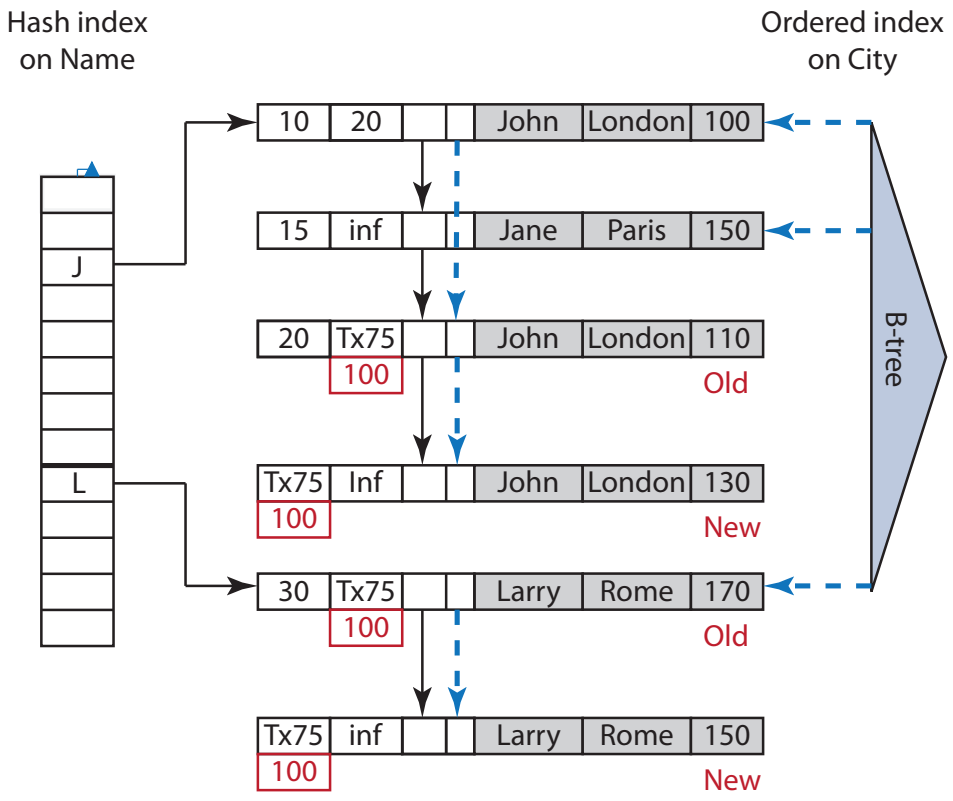
Indices:

- hash-table on name (here: hash first character)
- Bw-Tree on city

SQL Server: Hekaton: Storage Layer: example



Record format



SQL Server: Hekaton: Storage Layer: example

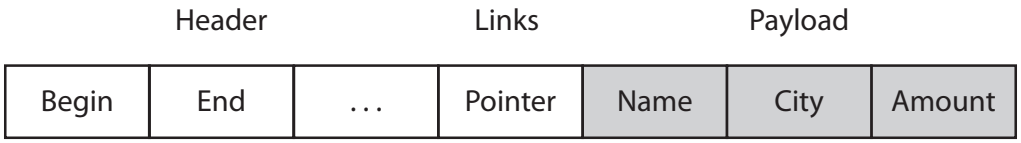
READ:

- reading is performed for a specific time
- for any time only one version of a record qualifies

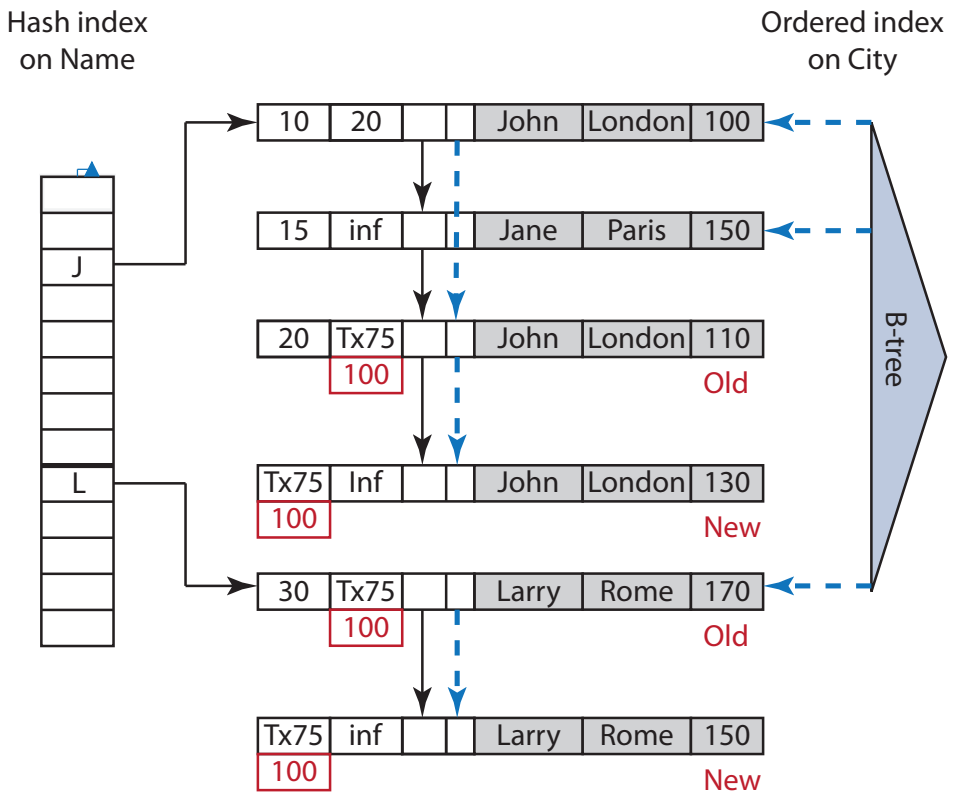
Update (red):

- TA 75 transfers 20 Yen from Larry's account to John's
- creates new versions of each account
- old version: 75 as their end-timestamp
- new versions: 75 in their begin-timestamps
- At commit time: update timestamps to commit time (100)

SQL Server: Hekaton: Storage Layer: example



Record format



Physical Algebra: Processing Modes

Physical Algebra: Processing Model

We split the discussion of the physical algebra into two parts:

1. Processing Modes
2. Implementation

We defer the discussion of the implementation after the discussion of expression evaluation since all operators need expression evaluation (e.g. a selection needs a selection predicate).

Physical Algebra: Pull

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

- open
- next
- close

`next` is called once per tuple.

Physical Algebra: Push

Producer interface:

- run

Consumer interface:

- init
- step
- fin

`step` is called once per tuple.

Physical Algebra: Push: 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;
};
```

Physical Algebra: Push: select

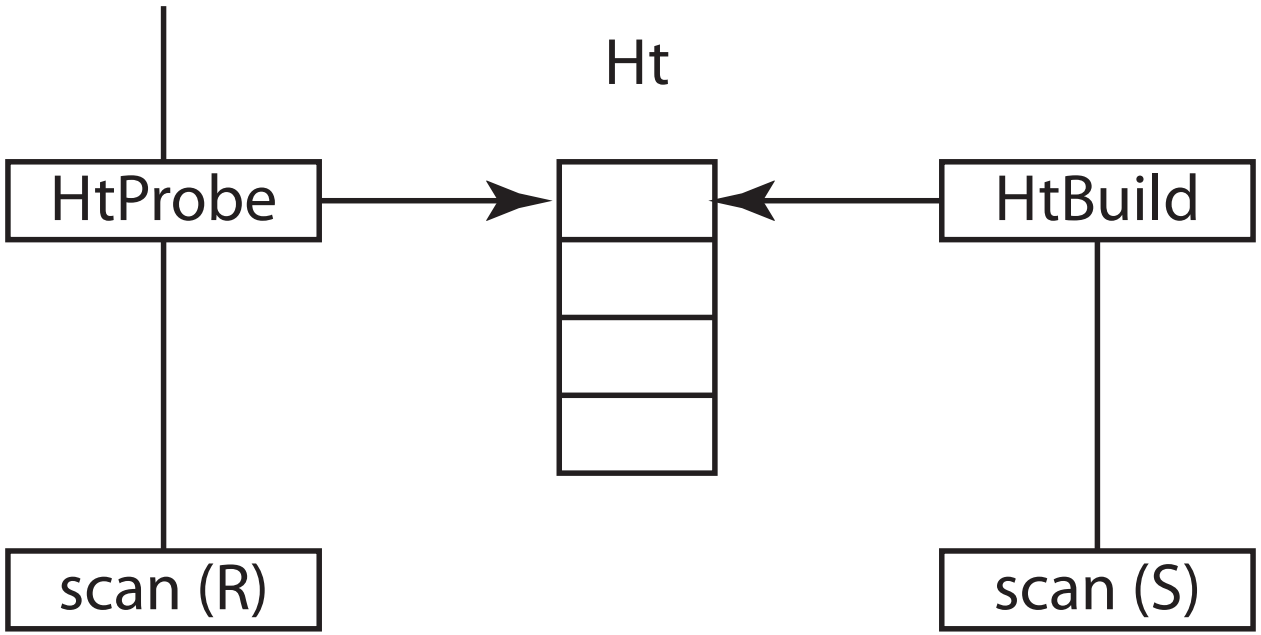
Sample code for selection:

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

Physical Algebra: Push: hash join (1)

The hash-join is split into two parts:

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



Physical Algebra: Push: hash join (2)

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

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

Physical Algebra: Push: hash join (3)

Pseudocode: For simplicity, we assume that

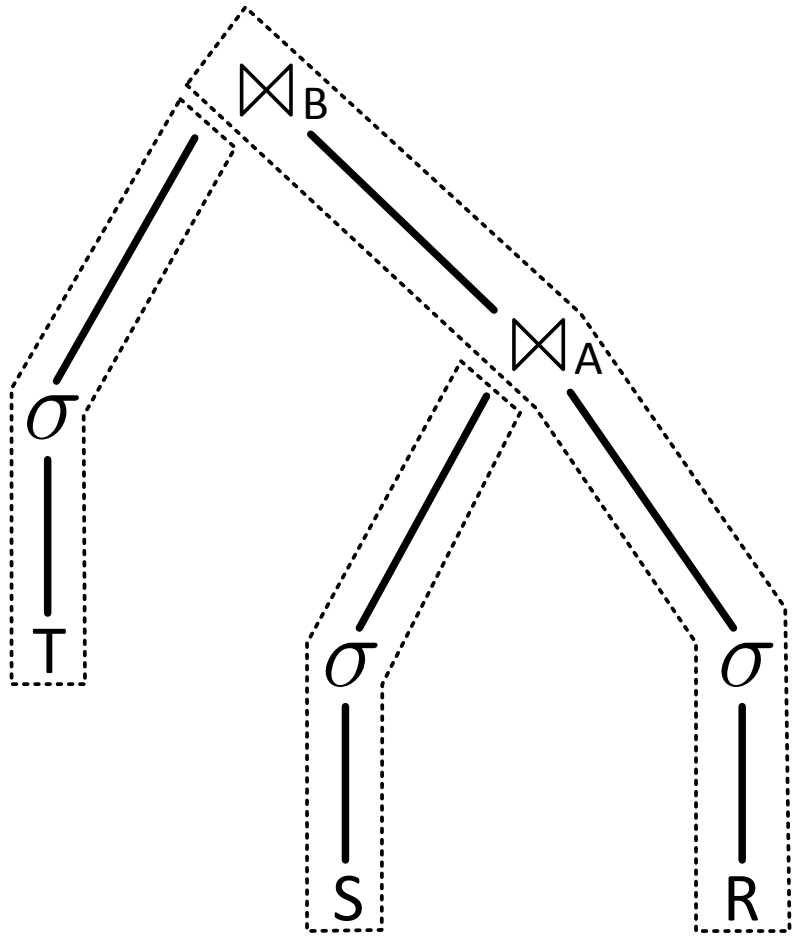
- the argument to the step function is a rid (i.e., the type of Tuple is uint) and every function knows how to access the right parts of the tuple.
- the hash functions h_r and h_s are somehow known and return an unsigned int (uint)
- the hash functions h_r and h_s take a rid as argument and implicitly 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.

Physical Algebra: Push: hash join (4)

```
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;
}
```

Physical Algebra: Push: Strands

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:



Physical Algebra: Processing Models

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)

Physical Algebra: Materialization Granularity: Single Tuple

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)).

Physical Algebra: Materialization Granularity: Full

- 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.

Physical Algebra: Materialization Granularity: Chunk/Vector

- 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).
- size of a chunk:
  - in bytes
  - in number of tuples

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

- one input bunch of tuples produces one output bunch of tuples.
- many input bunches of tuples can produce one output bunch of tuples.

Expression Evaluation

Expression Evaluation: Single Operator Xprs

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

Expression Evaluation: Multi Op Xprs

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.

Expression Evaluation: Possibilities

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
  - machine code

Expression Evaluation: 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)

Expression Evaluation: 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];
}
```

Expression Evaluation: Uvals

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;
```

Xpr Eval: 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
};
```

Xpr Eval: AVM: Program

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;
```

Xpr Eval: AVM: row: single tuple

Two general approaches: switch vs. function pointers.
In both cases, the signature is the same:

- return value: bool
- parameter:
  1. byte_t* `__restrict__` aTuple
  2. uint32_t* `__restrict__` aProg

Xpr Eval: AVM: single tuple: switch

```
#define OP(a1, a2, a3, op, T) (*(T*)(a1)) = (*(T*)(a2)) op (*(T*)(a3))
int
avm_itp_row_single_switch(byte_t* t, uint32_t* p) {
    int IRes = 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;
}
```

Xpr Eval: AVM: single tuple: funptr

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.

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

Xpr Eval: AVM: single tuple: funptr

```
int
avm_itp_row_single_funptr(byte_t* aTuple, uint32_t* p) {
    int IRes = 0;
    byte_t* t = aTuple;
    int IOp = 0;
    LOOP:
        IOp = *p++;
        if(kAvmStop == IOp) {
            goto END;
        }
        IRes = (gOpFunArr[IOp])((t + *p), (t + *(p+1)), (t + *(p+2)));
        p += 3;
        goto LOOP;
    END:
    return IRes;
}
```

Xpr Eval: AVM: row: vectorized

Above interpreter:

- per tuple
  - one call to AVM interpreter
  - per instruction in program
    - one branch/function call

Idea: reduce overhead by amortizing it on many tuples.
subsequently:

t	tuple
w	tuple width
n	number of tuples
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;
                    a2 += w;
                    a3 += w;
                }
                break;
            ...
        }
    goto LOOP;
    END:
    return n;
```

Xpr Eval: AVM: col: single

```
int avm_itp_col_single(byte_t* aColPtrs[], int aTupleNo, int* p) {
        int IRes = 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 IRes;
}
```

Xpr Eval: AVM: col: vectorized

```
int
avm_itp_col_vectorized(BYTE* 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(int32_t);
                a2 += sizeof(int32_t);
                a3 += sizeof(int32_t);
            }
            break;
```

Xpr Eval: AVM: col: vectorized: 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.

Xpr Eval: 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

Xpr Eval: Evaluation

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

$$A[0] = A[0] + A[1] - A[2] + A[3] - A[4]$$

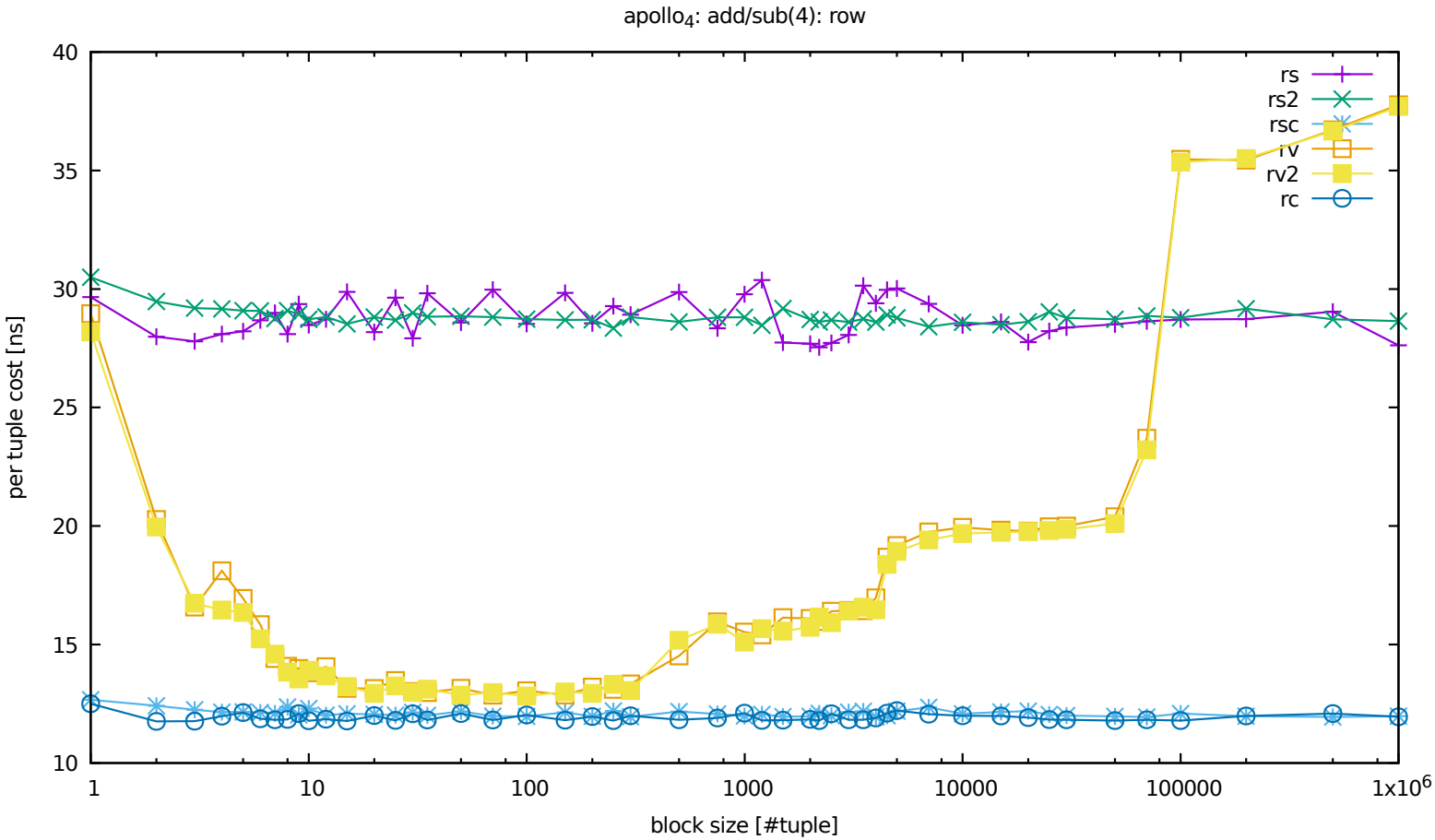
where $A[i]$ denotes the i-th integer attribute. The relation contained a total of 90 integer attributes and no other ones.

Xpr Eval: Evaluation

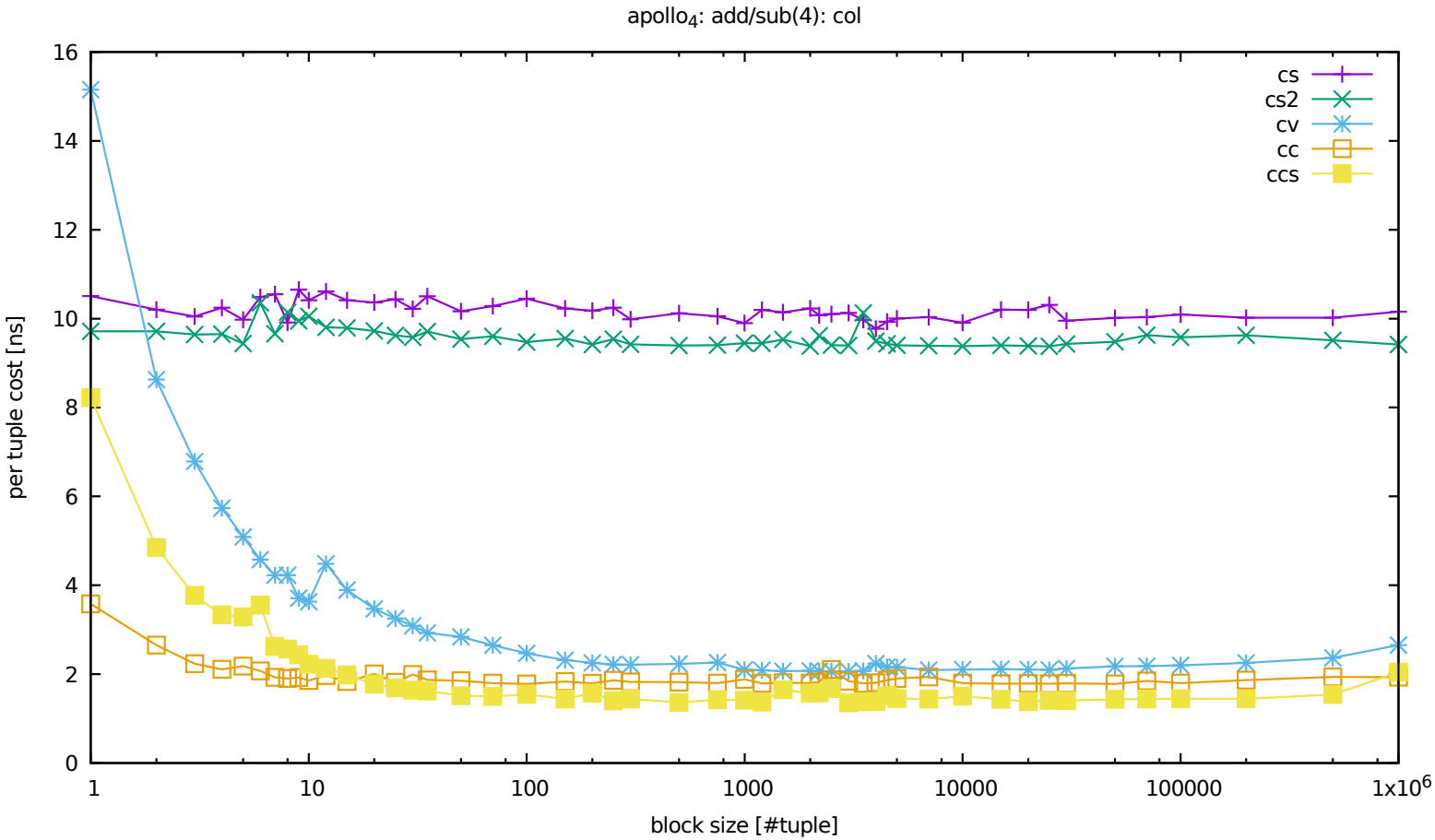
rs,rs2	row single interpreted switch/function pointer
rsc	row single compiled
rv,rv2	row vectorized interpreted, two slightly varying implementations
rc	row vectorized compiled
cs,cs2	col single interpreted switch/function pointer
cv	col interpreted vectorized
cc	col compiled vectorized, without SIMD
ccs	col compiled vectorized, with SIMD

[SIMD instructions generated by compiler]

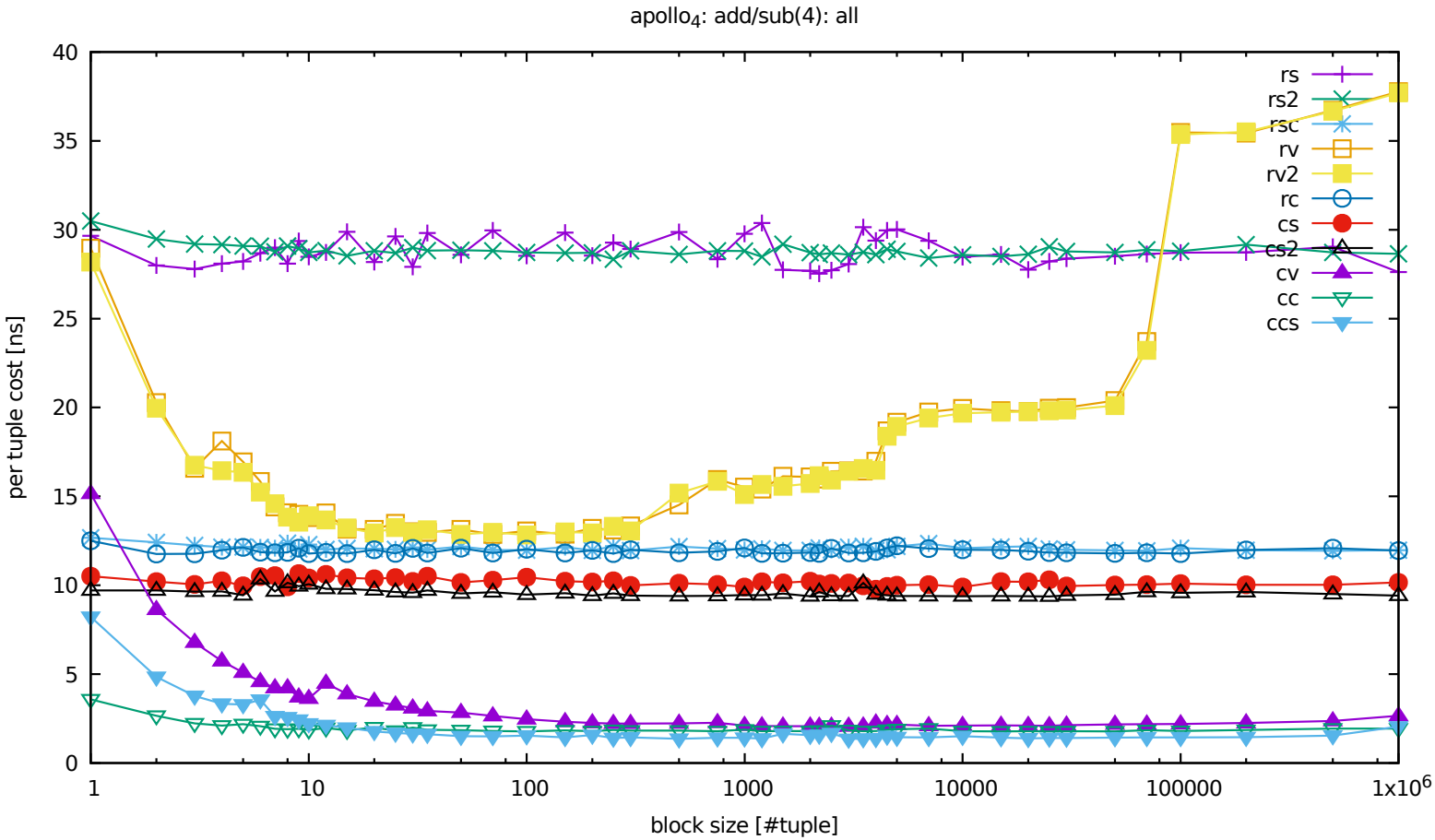
Xpr Eval: Eval: Row



Xpr Eval: Eval: Col



Xpr Eval: Eval: Row and Col



Xpr Eval: 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.

Xpr Eval: AVM: col: Vectorized SIMD: selection

In order to avoid using special SIMD-instruction, which makes the code somewhat more difficult to read, we use the following notation:

- W is the number of entries in one SIMD-register.
  For example: 4 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.

Xpr Eval: AVM: col: Vectorized SIMD: selection

The algorithm needs/does:

- performs a selection with a `between` predicate on some key column
- some input column T_{in} containing the key attribute
- some input column P_{in} containing some payload
- 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}.
- a software-managed buffer B
- a index vector $\vec{r}$ containing the current row ids.

SELECT_BETWEEN

$i, j, l \leftarrow 0$ // index for in/out/buffer
$\vec{r} \leftarrow [0, \ldots, W-1]$ // input indices
for($i = 0$, $i < |T_{in}|$; $i += W$) // for each lane
  $\vec{k} \leftarrow T_{in}[i]$ // read W input values
  $m \leftarrow (\vec{k_{lb}} \leq \vec{k}) \& (\vec{k} \leq \vec{k_{ub}})$ // 'between' to mask
  if($0 \neq m$) // at least one qualifying input key?
    $B[l] \leftarrow_m \vec{r}$ // selectively store indices
    $l \leftarrow l + |m|$ // inc each component by $|m|$
    if($|B| - W < l$) // buffer almost full?
      for($b = 0$; $b < |B| - W$; $b += W$) // step through buffer
        $\vec{x} \leftarrow B[b]$ // load idx of qualifying tuples
        $\vec{k} \leftarrow T_{\text{in}}[\vec{x}]$ // load qualifying keys
        $\vec{p} \leftarrow P_{\text{in}}[\vec{x}]$ // load qualifying payload
        $T_{\text{out}}[j + b] \leftarrow \vec{k}$ // store key values
        $P_{\text{out}}[j + b] \leftarrow \vec{p}$ // store payload
      $\vec{p} \leftarrow B[|B| - W]$ // move overflow ..
      $B[0] \leftarrow \vec{p}$ // .. to buffer begin
      $j \leftarrow j + |B| - W$ // update output index
      $l \leftarrow l - |B| + W$ // update buffer index
  $\vec{r} \leftarrow \vec{r} + W$ // update index vector
// after loop: flush remaining items in buffer

Physical Algebra

Physical Algebra

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

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

Physical Algebra: Techniques

1. Blocking/Tiling
2. Partitioning
3. Extraction
4. Loop Fusion
5. software managed buffers
6. explicit prefetching
7. streaming stores (possibly with software write-combining)

Physical Algebra: Techniques: Blocking/Tiling

Nested loop join like algorithm:

- each element from one input is compared to each element with some other input.

Inputs: arrays X and Y.

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

Can be rewritten to

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

where B is the block-size, such that B elements of Y fit into the cache.

Physical Algebra: Techniques: 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

Physical Algebra: Techniques: Extraction

Instead of sorting full tuples or inserting full tuples into a hash table, we can use

- pairs of sort-key and pointers to tuples

or similar (hash-key, hash-value, pointer/rid/tid).

Physical Algebra: Techniques: Loop Fusion

Extraction and hash table insert implemented with two 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])
```

here: most probably $A[i]$ in cache

Physical Algebra: Operator Overview

Overview:

1. scan/select
2. join
3. partitioning
4. sorting (*)
5. grouping/aggregation (*)

(*): not yet

Physical Algebra: Scan/Select

We have discussed most alternatives already:

- branching code versus predicated code
- SIMD

Physical Algebra: Join: Simple

Start: simple hash join ($S \bowtie^{hj} R$):

HtBuild(H_R, R)
for each $s \in S$
    Probe(s, H_R)

Discussion:

- 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.

Physical Algebra: Join: Extraction

Improvement: extract key-pointer-pairs from R:

for each $r \in R$
  H_R.insert(ExtractKeyPointer(r))
for each $s \in S$
  Probe(s, H_R)

Discussion:

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

Physical Algebra: Join: Partitioning

Partition both relations:

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

Partitioning details: next section.

Physical Algebra: Join: Software Prefetching

- software prefetching is an alternative to partitioning.
- three techniques
  - group prefetching
  - software-pipelined prefetching
  - rolling prefetching

Physical Algebra: Join: group prefetching

Probe:

```
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]

Physical Algebra: Join: group prefetching

Disadvantages of group-prefetching:

1. bursts of prefetches
2. complexity

Physical Algebra: Join: Software-Pipelined Prefetching

Probe (D = pipeline length):

```
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

Physical Algebra: Join: Rolling Prefetching

Parameter $k = 2$:

```
template<class Tuint, class Tbun, class Thashfun>
void build_rp_2(const std::vector<Tbun>& aBun) {
    const size_t m = size(); HtSize
    const size_t n = aBun.size();
    Tuint IIdxA = 0; // number 1
    Tuint IIdxB = 0; // number 2 (=k)
```

```
    if(2 < n) {
        IIdxA = Thashfun()(aBun[0].key()) % m;
        IIdxB = Thashfun()(aBun[1].key()) % m;
        __builtin_prefetch(&(_dir[IIdxA]), 1, 0); // optional
        __builtin_prefetch(&(_dir[IIdxB]), 1, 0); // optional
        const size_t nx = n - 2;
        for(size_t i = 0; i < nx; ++i) {
            insert_at(aBun[i], IIdxA);
            IIdxA = IIdxB;
            IIdxB = Thashfun()(aBun[i+2].key()) % m;
            __builtin_prefetch(&(_dir[IIdxB]), 1, 0);
        }
        for(size_t i = nx; i < n; ++i) {
            insert(aBun[i]); // process the rest
        }
    } else {
        build(aBun); // simple build for small relations
    }
}
```

Physical Algebra: Join: Rolling Prefetch

- The parameter k determines the distance between the hash directory entry currently inserted into and the hash directory entry currently prefetched
- $k = 2$ does not allow for sufficient work inbetween to hide memory access latency
- increase k by adding IIdxC, IIdxD, etc. is a little cumbersome.
- next: code for $k = 8$ with array instead of single variables

Physical Algebra: Join: Rolling Prefetch

```
template<class Tuint, class Tbun, class Thashfun>
void
build_rp_8(const std::vector<Tbun>& aBun) {
    const size_t m = size(); // HtSize
    const size_t n = aBun.size(); // input size
    Tuint lIdx[8]; // eight temporal variables, used round robin
    const uint32_t lMask = 0x7; // mask for round robin
```

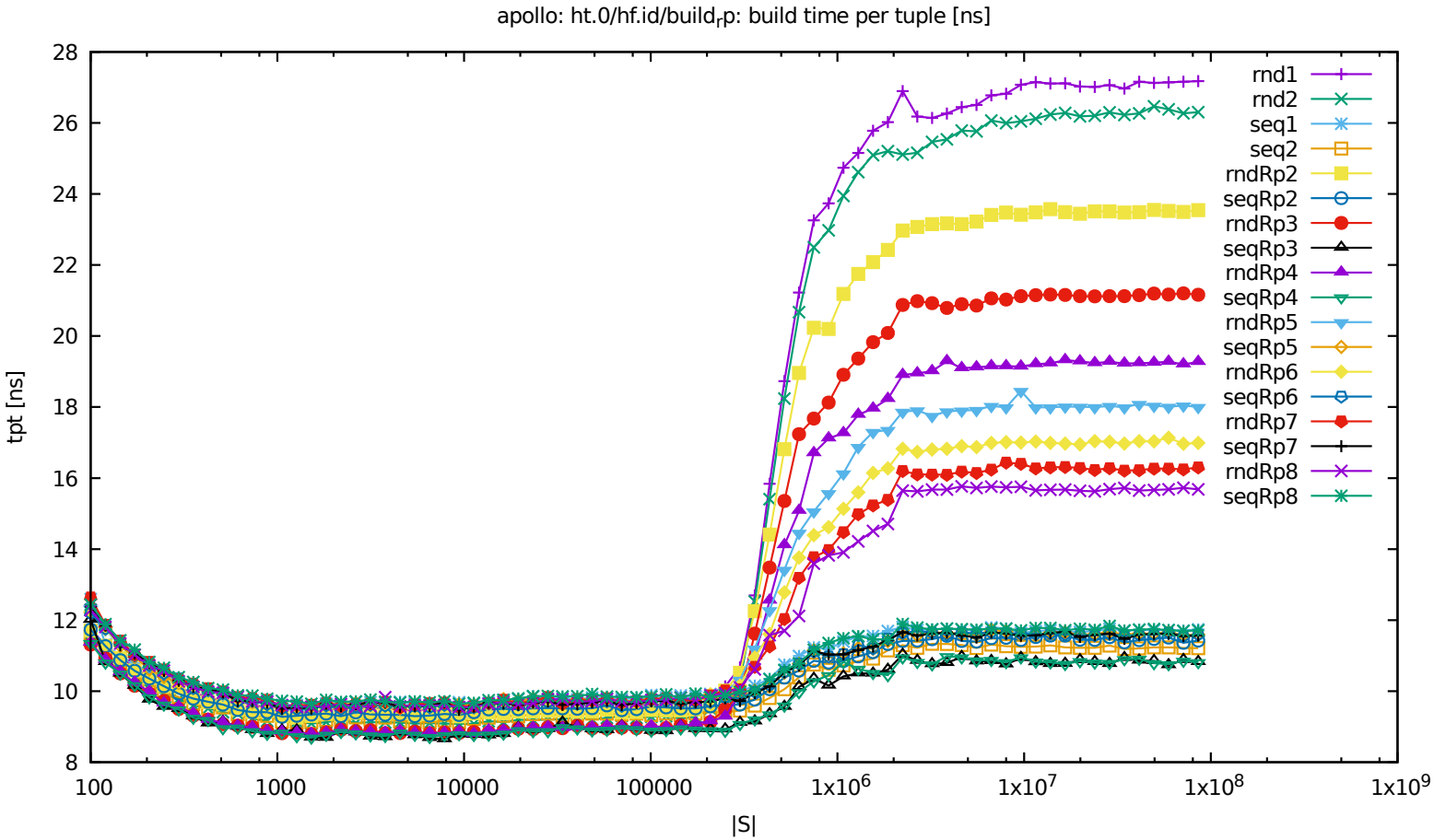
```
    if(8 < n) {
        for(int i = 0; i < 8; ++i) {
            lIdx[i] = Thashfun()(aBun[i].key()) % m;
            __builtin_prefetch(&(_dir[lIdx[i]]), 1, 0);
        }
        const size_t nx = n - 8;
        uint32_t lCurr = 0;
        for(size_t i = 0; i < nx; ++i, ++lCurr) {
            insert_at(aBun[i],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

Physical Algebra: Join: Rolling Prefetch: Performance

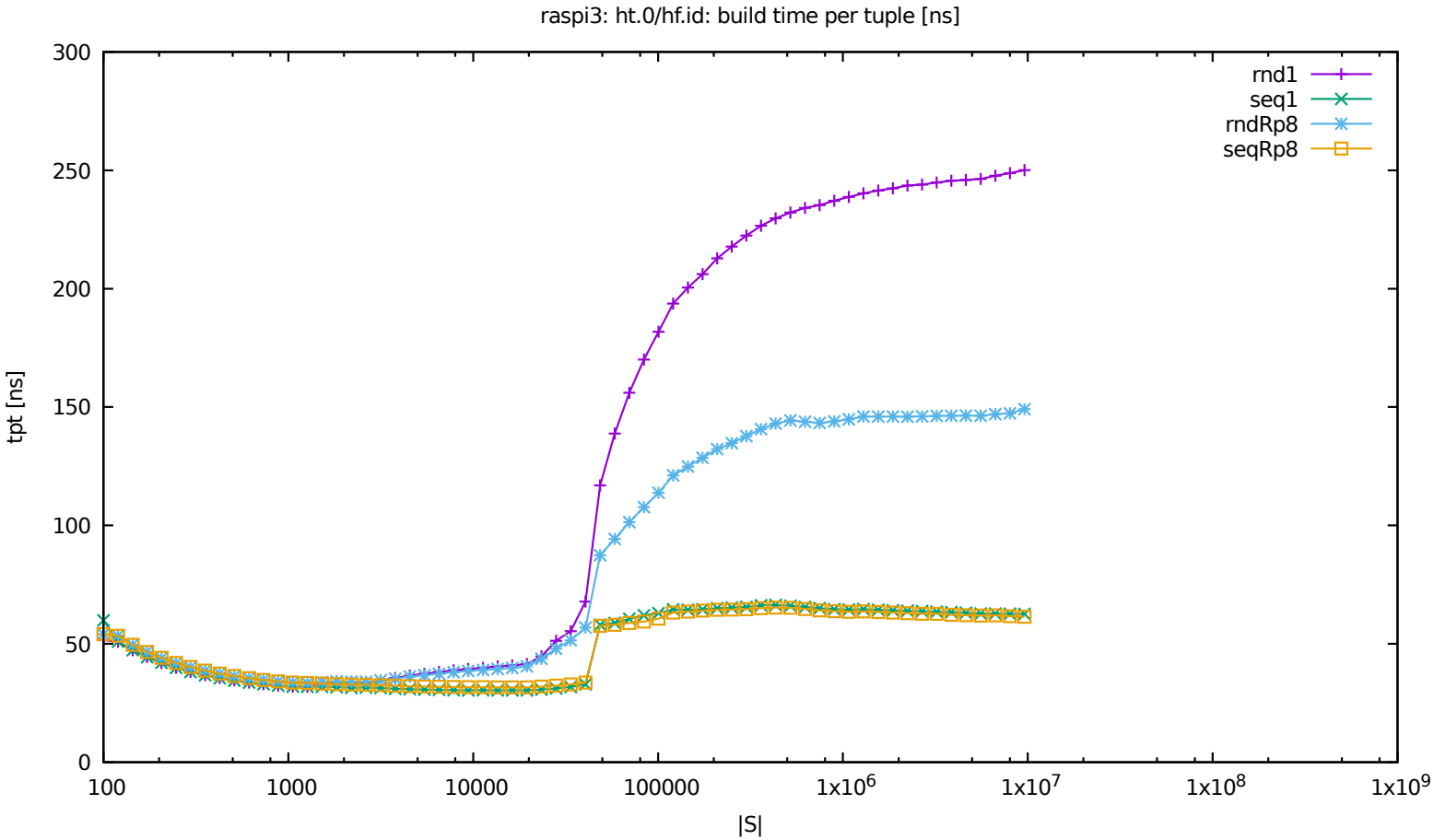
performance test:

- cheap hash function: identity
- on sorted (seq) and randomly permuted (rnd) key
- evaluate
  - simple hash build
  - rolling prefetch build: vary parameter k from 2 to 8
- x-axis: cardinality of build input
- y-axis: time per build tuple

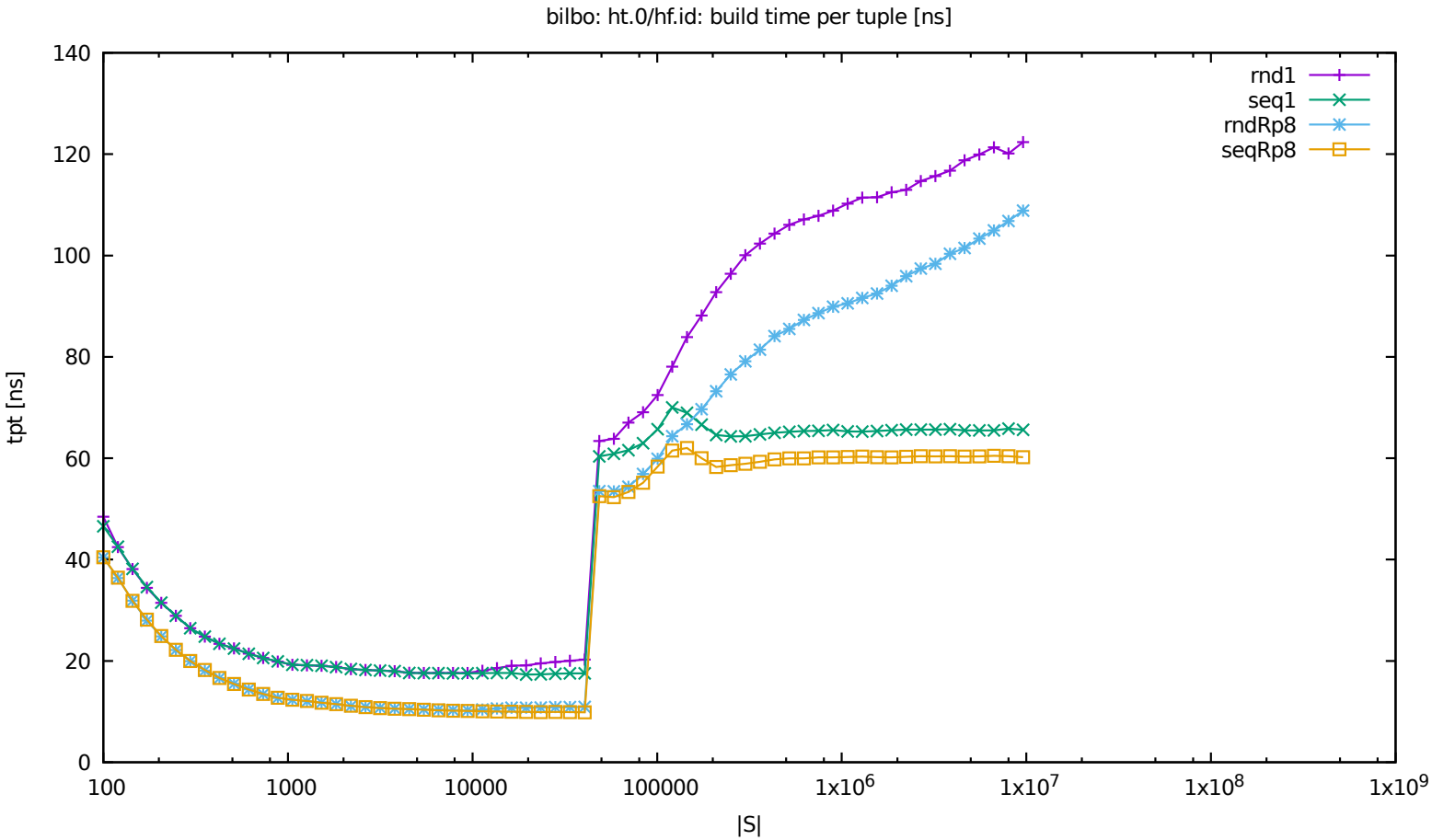
Physical Algebra: Join: Build: i7-4790



Physical Algebra: Join: Build: Raspberry Pi 3



Physical Algebra: Join: Build: XU-4



Rolling Prefetching: Discussion

- easy to implement
- only prefetches directory entries
- does not prefetch collision chain entries

Asynchronous Memory Access Chaining (AMAC)

Main Idea:

- keep address and
- execution state in a small
- array organized as a
- ring buffer

Probe: state: distinguish between

- hashing/prefetching and
- subsequent comparison/access

Assumption here: hash directory entry and collision chain element have the same structure. Otherwise another code fragment (and thus state) must be introduced.

AMAC: state

```
struct state_t {
    uint64_t  idx;      // index/rid of current input element
    uint64_t  key;      // key of the current input element
    uint64_t  pload;    // payload of the current input element
    node_t*   ptr;      // hash directory entry or collision chain elem
    int32_t   stage;    // handle hash dir entry or collision chain elem
};
```

AMAC: probe: part I

```
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) { // collision chain element
            entry_t* n = s[k].ptr;
            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) */
            }
```

AMAC: probe: part II

```
        } 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 */
}
```

AMAC: discussion

- fully handles all cases
- can be applied to other algorithms
- introduces sequence of if-statements or switch (branch misprediction!) to hide main memory access latency

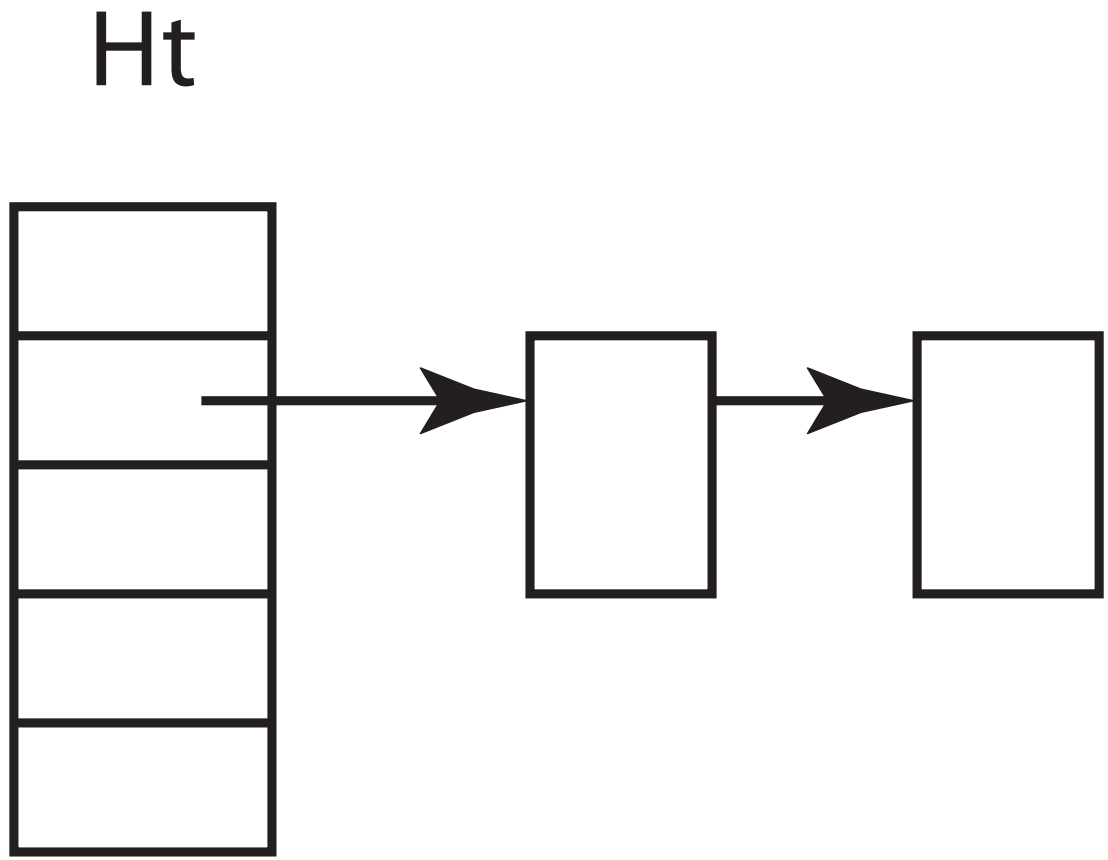
Partitioning

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.
- A goal of partitioning is to store items in the partitions in close neighborhood, i.e., *clustered*.

Partitioning: Simple

A simple hashtable is used to point to the partitions which are allocated in chunks and possibly chained.



Partitioning: Simple and Radix

```
#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`

also: the code

- contains two comments concerning some optimization potential.
- is more complex since it can be used in multiple passes useful if
  - 2^D pointers are larger than Ld1/2/3, TLB1/2.
  - 2^D exceeds the number of TLB1/2 entries.

Multi-Pass Radix-Partitioning

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

$\Longrightarrow$ 1*lsb*

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

msb: most significant bit, lsb: least significant bit

Partitioning: Why chunks are not so good

One problem with the approach of having chained output chunks is a possible

- underutilization of memory as some chunks maybe partially filled.

Idea:

- instead of chunks
- use densely populated array to store partitions

Partitioning: dense array

- idea: use histogram to determine offset of partitions within a densely packed output array
- subsequently: f is the function used for partitioning

Partitioning: Histogram Build

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)]++;
}

Partitioning: Histogram Prefix Sums Are Offsets

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];
    }
}
```

Partitioning: cache-oblivious

```
part0(S, O, T) {
    for(int i = 0; i < |T|; ++i) {
        t = T[i]; // get input tuple t
        off = O[f(t.key)] + +; // get 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.

Partitioning: cache-oblivious: in-place

- For multiple passes, in-place partitioning might be useful.
- For this algorithm we need the end of each partition

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

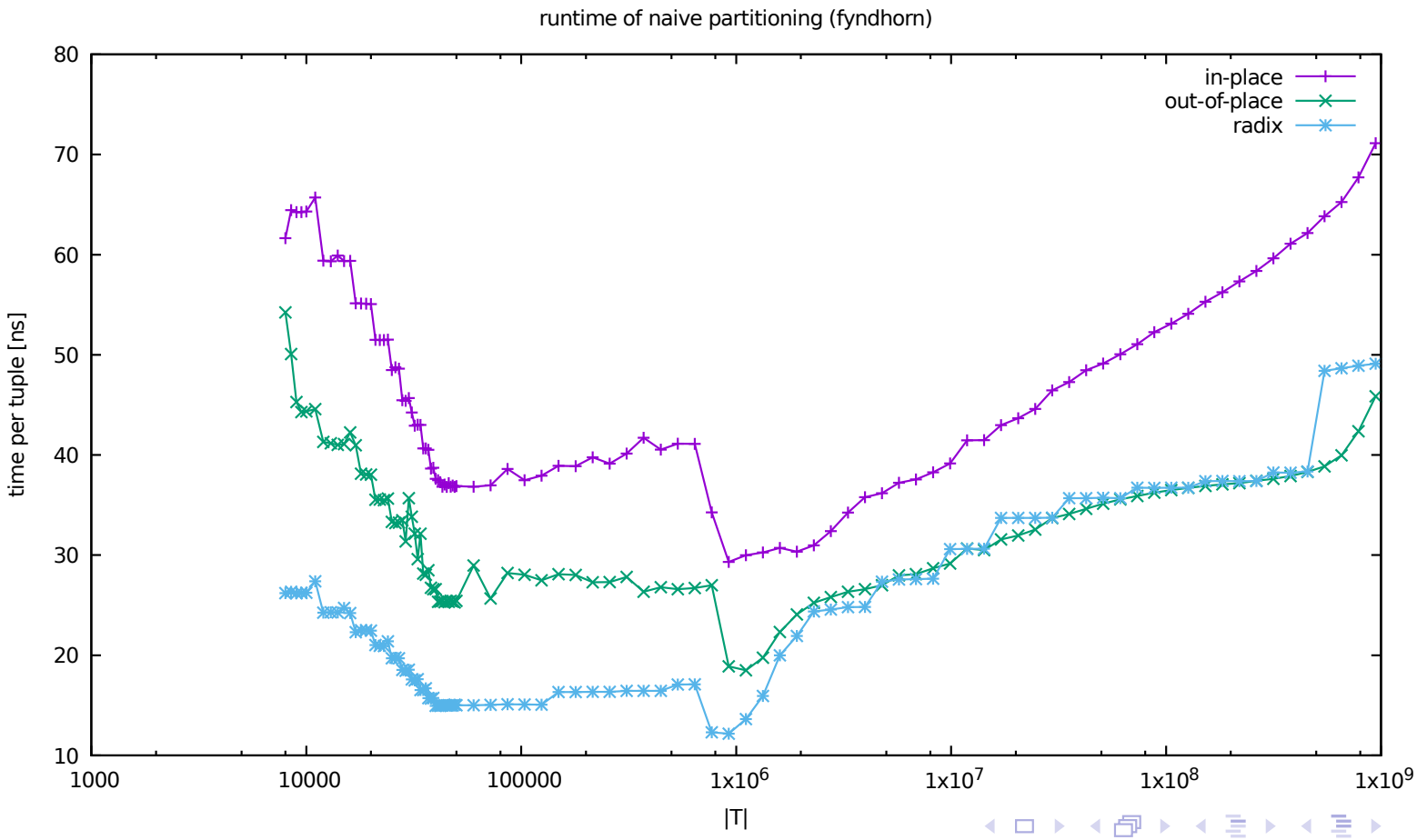
Partitioning: cache-oblivious: in-place

T: input and output table; H: is the histogram; O: offset array produced by `offset_end`; P: number of partitions.

```
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++];
        } while((p < P) && (i == O[p]));
    } while(p < P);
}
```

Partitioning: runtime

The following figure shows the runtime of the out-of-place, in-place, and radix-cluster algorithms. 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).



Partitioning: software-managed buffer

The following code

- uses the last entry in the buffer to store the current offset of a partition

This avoids another cache miss.

Partitioning: software-managed buffer

```
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;
        }
    }
}
```

Physical Algebra: Sort

not this semester

Physical Algebra: Grouping and Aggregation

not this semester

Index structures

Index Structures: Cache Conscious B$^+$-Tree

The main idea of the CSB$^+$-Tree:

- Instead of $k + 1$ child pointers for k keys, the CSB$^+$-Tree stores only one or a few child points.
- One child pointer suffices if successive child nodes are stored consecutively in memory.
- In its simplest variant (full CSB$^+$-Tree), there is always (!) space allocated for the maximum number of child nodes.

As usual:

- A CSB$^+$-Tree of order d contains k keys with $d \leq k \leq 2d$.

Index: CSB$^+$-Tree: inner node

```
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
}
```

The size of a node here is 64 byte, which is exactly one cache line. In general, a node can comprise multiple (a few) cache lines.

Index: CSB$^+$-Tree: child node allocation

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.

Index: CSB$^+$-Tree: leaf nodes

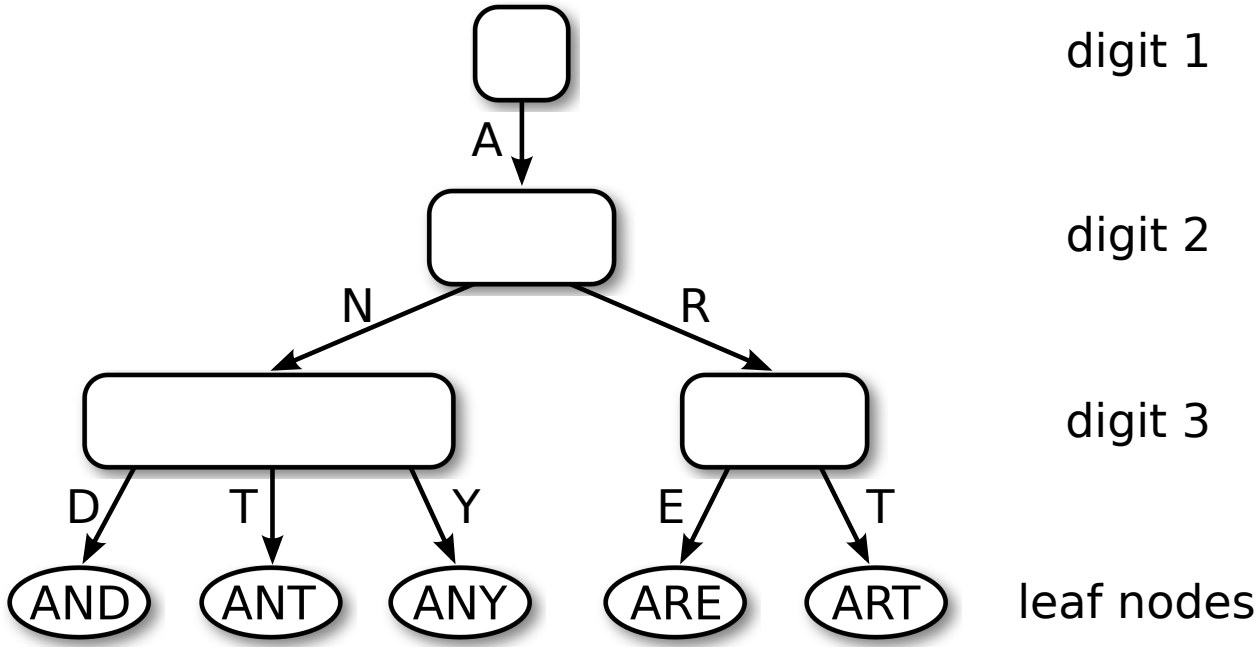
Leaf nodes contain (key,ptr/rid)-pairs and are chained:

- first sibling of a node group contains `previous` pointer
- last sibling of a node group contains `next` pointer
- otherwise offset calculation is used

Index: CSB$^+$-Tree: operations

The operations in the CSB$^+$-Tree are very similar to those in a regular B^+-Tree.

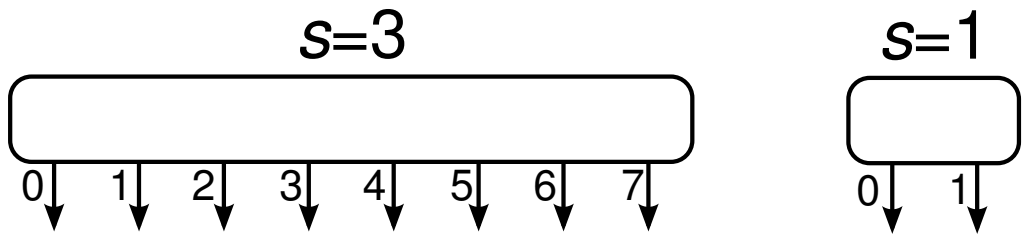
Index: Remember Radix Trees (TRIE)



- Tree height depends on key length k, but not on tree size n
- No re-balancing required
- Lexicographic order
- The keys stored implicitly, reconstructable from paths

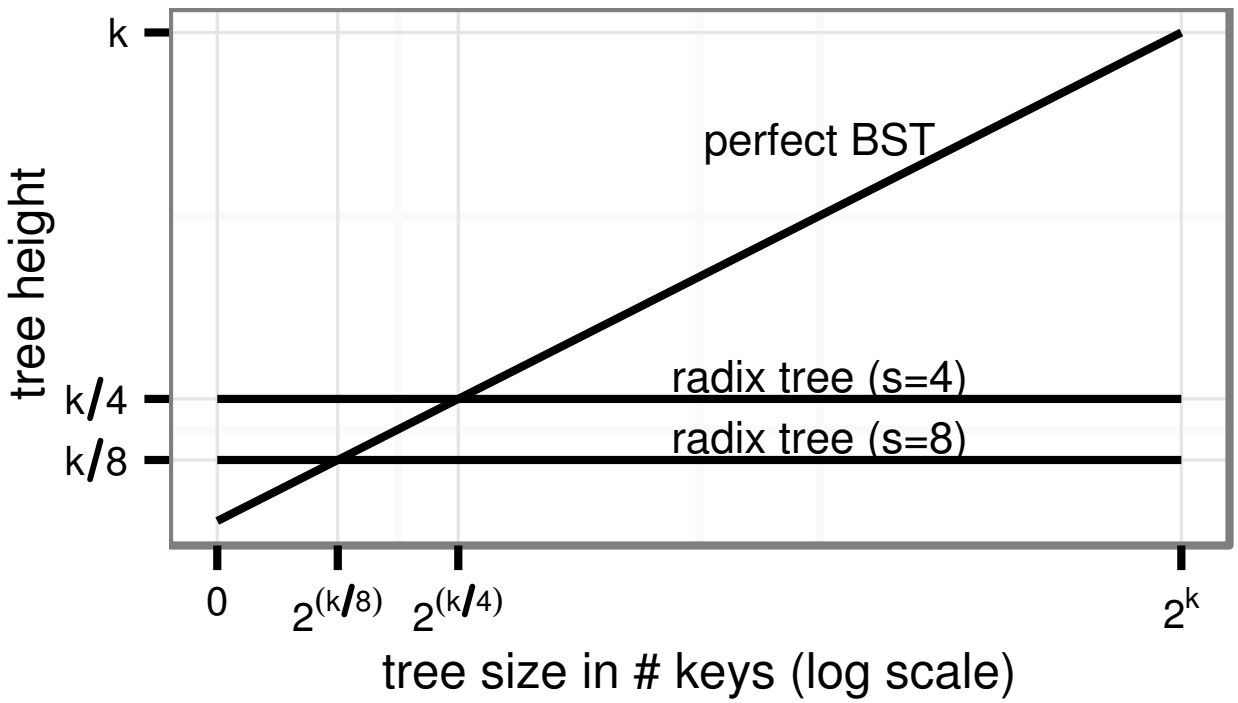
Index: Radix Tree

- For binary keys, the fanout can be configured.
- At each node, s bits ("span") of the key are used.
- Each inner node is simply an array of 2^s pointers.



Index: ART: Adaptive Radix Tree

Why radix tree and not balanced binary search tree?
Height of a perfectly balanced binary search tree and a radix tree:



Index: ART: Adaptive Radix Tree

Traditional inner node of a radix tree:

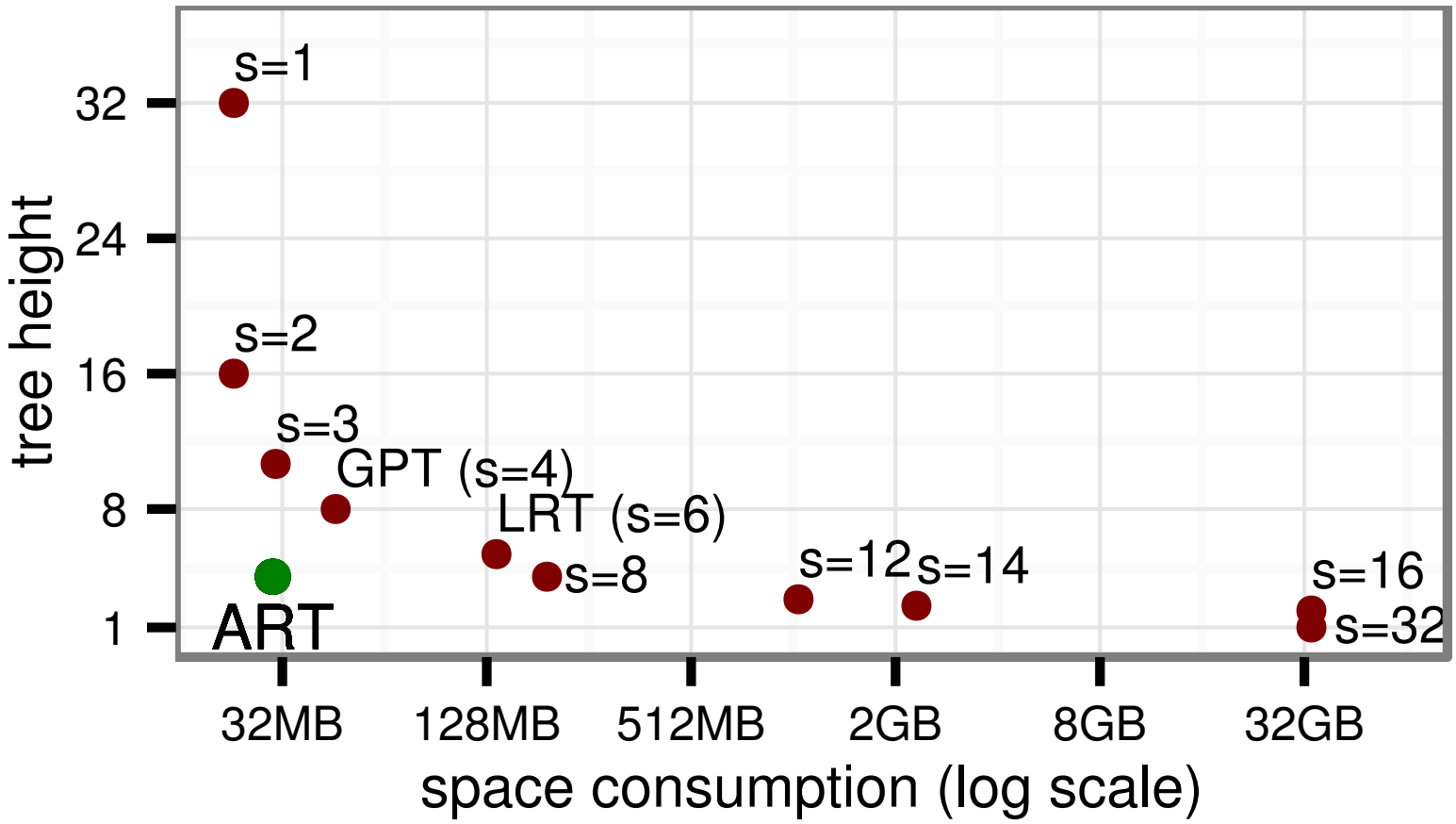
- 2^s pointers

for a *span* of s bits of the key.

- If the key is k bits long, the radix tree has height $\lceil k/s \rceil$.
- Thus, s is a critical parameter for radix tree height.
- Also: s is a critical parameter for radix tree space consumption.

Index: ART: Space Consumption

s is critical for height and space usage:

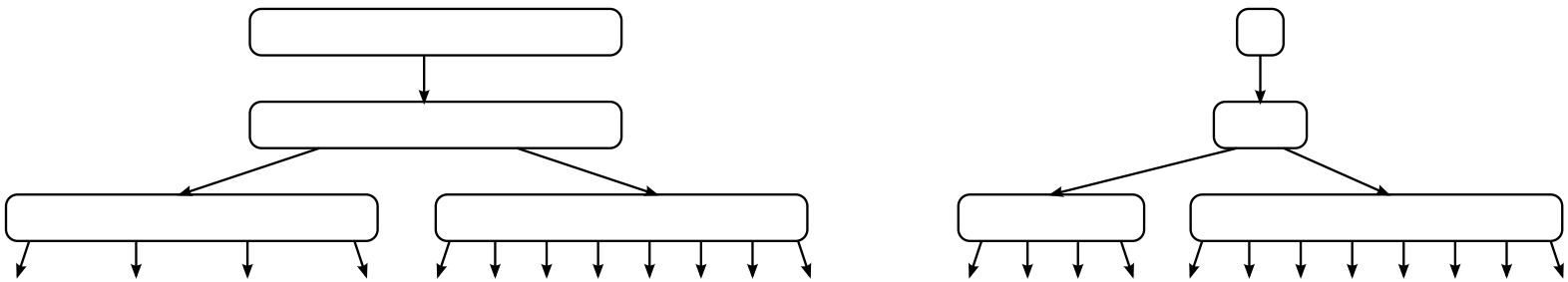


Index: ART: influences of s

- only some choices for s are suitable:
  - the larger s the better the lookup performance
  - the smaller s the smaller the space consumption
- ART: reduction of space due to multiple node sizes (see next slide)

Index: ART: Adaptive Radix Tree

Problem in regular radix tree: partially filled nodes (left):

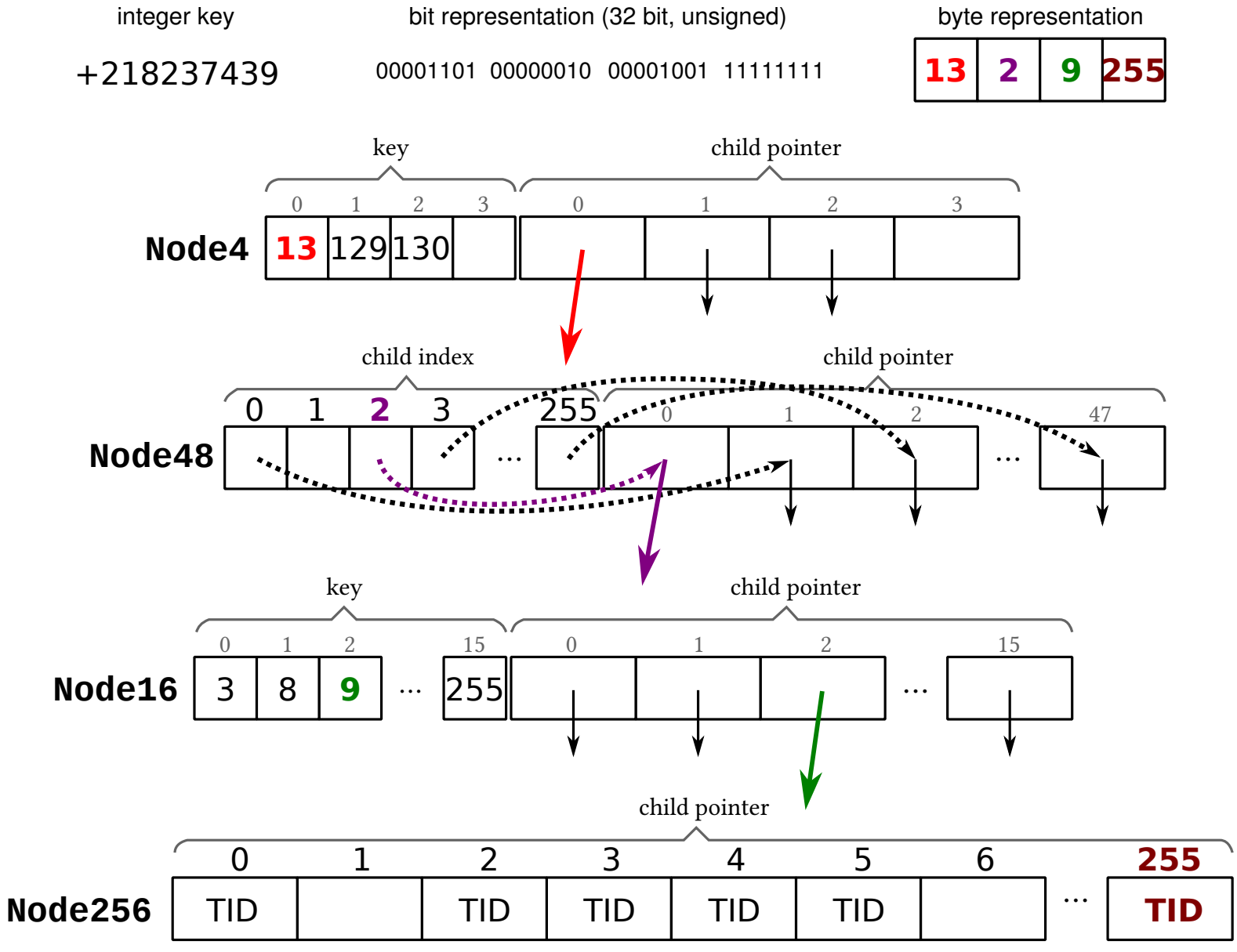


ART: different node sizes (right)

Index: ART: inner nodes

Node4	stores up to 4 child node pointers and up to 4 keys
Node16	stores between 5 and 16 child node pointers and keys
Node48	stores stores an array with 17 to 48 child node pointers and 255 offsets into this array
Node256	stores an array of 256 entries.

All nodes have a header containing node type, number of child nodes, and compressed path to the node.


integer key
bit representation (32 bit, unsigned)
byte representation
+218237439
00001101 00000010 00001001 11111111
13 2 9 255
key
child pointer
Node4
0 1 2 3
13 129 130
child index
child pointer
Node48
0 1 2 3 255
0 1 2 47
key
child pointer
Node16
0 1 2 15
3 8 9 255
child pointer
Node256
0 1 2 3 4 5 6 255
TID TID TID TID TID TID

Index: ART: leaf nodes

Here: only unique index

1. single-value leaves: store one value
2. multi-valued leaves: store several key/value stores, may differ in structure as inner nodes
3. combined pointer/value slots: if values fit into pointers, e.g., `sizeof(void*)` >= `sizeof(TID)`
   one can reuse the inner node structures.

- single-value leaves most general, but increases key height (additional pointer chase)
- multi-valued leaves good [hier fehlt was] ???
- combined pointer/value slots: preferable mode of operation

Index: ART: height reduction

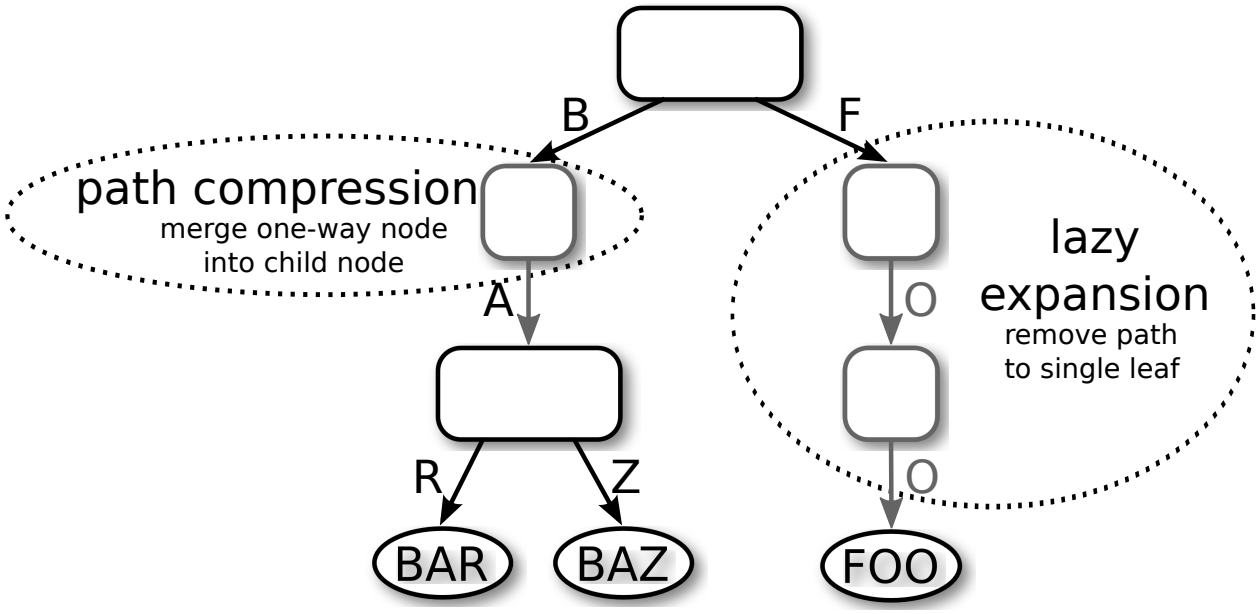
Long keys result in large height. Two techniques to reduce height:

lazy expansion inner nodes are only created if needed to distinguish two leaf nodes

path compression remove inner nodes with a single child only

The latter requires to store the 'left out' part of the key to be stored in the nodes. In ART: 8 bytes fixed. if exeeded: compare complete key in leaf nodes or after index at tuple access time.

Index: ART: height reduction



Index: ART: lookup

- `lookup` finds leaf by successively calling `findChild`
- `findChild` looks up a child in an inner node, given a partial path (one byte)

Index: ART: findChild (1)

```
findChild(node, byte)
    if node.type == kNode4 // simple loop
        for (i=0; i < node.count; ++i)
            if node.key[i] == byte
                return node.child[i]
        return NULL
    if node.type == kNode16 // use SIMD
        key = _mm_set1_epi8(byte)
        cmp = _mm_cmpeq_epi8(key, node.key)
        msk = (1 << node.count) - 1
        bv = _mm_movemask_epi8(cmp) & msk
        if bv
            return node.child[ctz(bv)]
        else
            return NULL
```

Index: ART: findChild (2)

```
if node.type == kNode48 // two array lookups
    if node.childIndex[byte] != kEmpty
        return node.child[node.childIndex[byte]]
    else
        return NULL
if node.type == kNode256 // if not really needed
    return node.child[byte] // single array lookup
```

Index: ART: insert (1)

we use the following subroutines:

- `replace` replaces a node in the tree by another node
- `addChild` appends a new child to an inner node
- `checkPrefix` compares the compressed path of a node with the key and returns the number of equal bytes
- `grow` replaces a node by a larger node
- `loadKey` retrieves the key of a leaf

Index: ART: insert(2)

```
insert(node, key, leaf, depth)
    // case 1: empty tree
    if node == NULL // handle empty tree case
        replace(node, leaf)
        return
```

Index: ART: insert(3)

```
insert(node, key, leaf, depth)
    ...
    // case 2: existing leaf is encountered
    // (possibly due to lazy expansion)
    if isLeaf(node) // expand node
        newNode = makeNode4()
        key2 = loadKey(node)
        for (i = depth; key[i] == key2[i]; ++i)
            newNode.prefix[i-depth] = key[i]
        newNode.prefixLen = i - depth;
        depth += newNode.prefixLen
        addChild(newNode, key[depth], leaf)
        addChild(newNode, key2[depth], node)
        replace(node, newNode)
        return
```

Index: ART: insert(4)

```
insert(node, key, leaf, depth)
    ...
    // case 3: key of the new leaf to be inserted
    // differs from compressed path
    p = checkPrefix(node, key, depth) // len common prefix
    if p != node.prefixLen // prefix mismatch
        newNode = makeNode4()
        addChild(newNode, key[depth+p], leaf)
        addChild(newNode, node.prefix[p], node)
        newNode.prefixLen = p
        memcpy(newNode.prefix, node.prefix, p)
        node.prefixLen = node.prefixLen - (p + 1)
        memmove(node.prefix, node.prefix + p + 1, node.prefixLen)
        replace(node, newNode)
        return
```

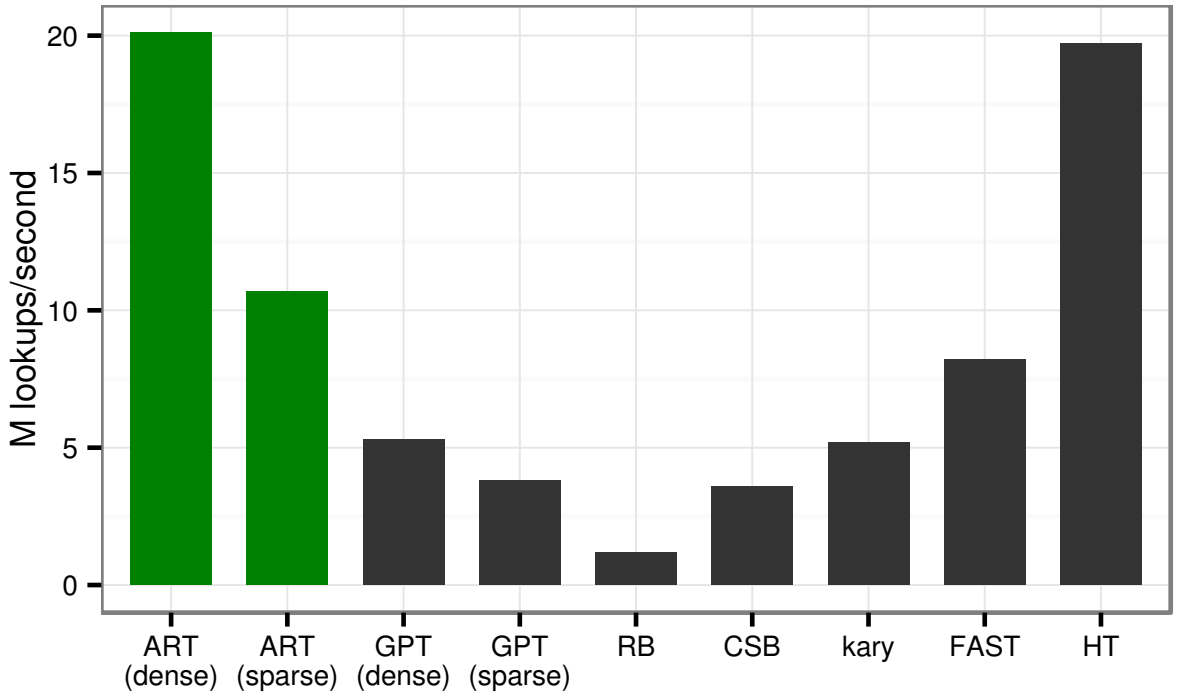
Index: ART: insert(5)

```
insert(node, key, leaf, depth)
    ...
    // case 4: regular cases
    depth += node.prefixLen
    next = findChild(node, key[depth])
    if next // recurse
        insert(next, key, leaf, depth + 1)
    else
        if isFull(node)
            grow(node)
        addChild(node, key[depth], leaf)
```

Index: ART: bulkload

- recursively partition data
- build ART accordingly

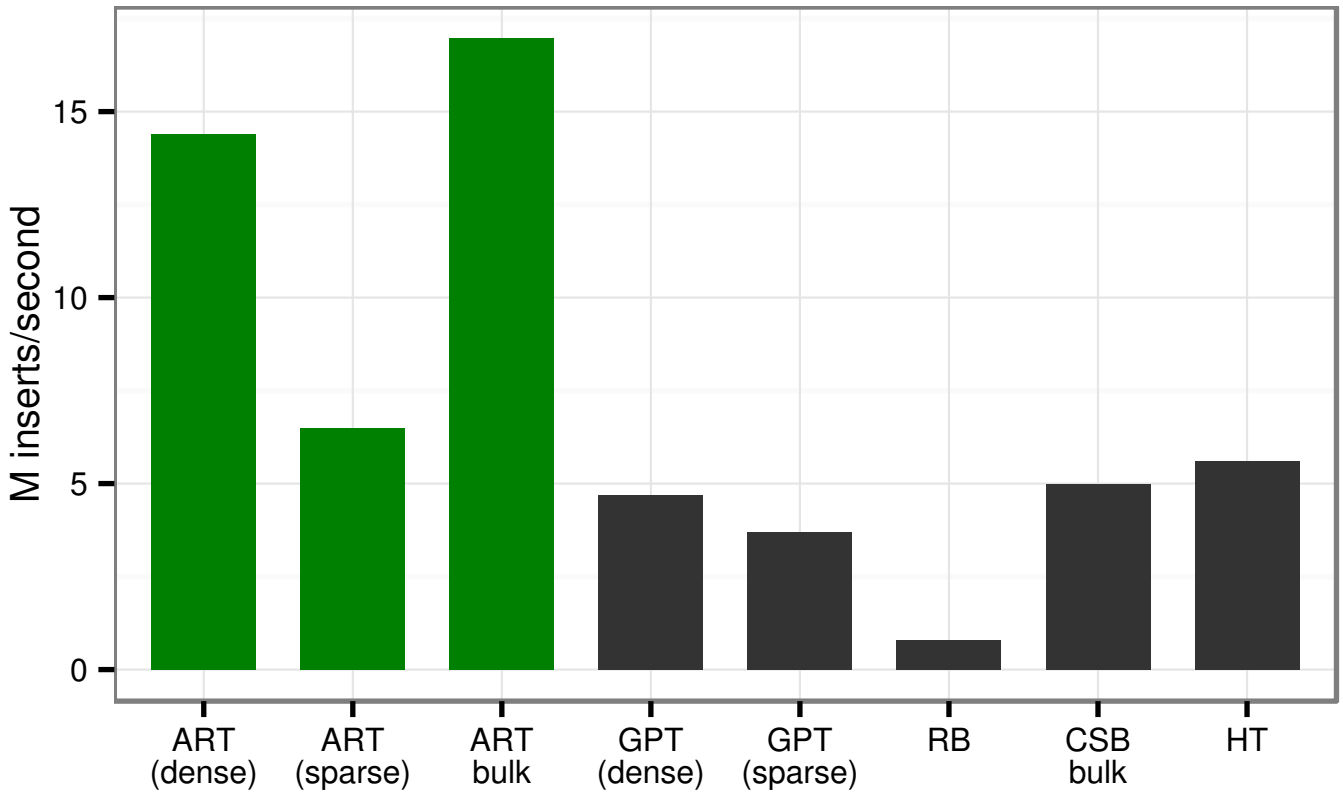
Index: ART: Performance: Lookup (4 Byte Keys)



- GPT: Generalized Prefix Tree, Boehm et al., BTW 2011
- RB: Red-Black Tree
- CSB: Cache-Sensitive B+Tree, Rao and Ross, SIGMOD 2000
- kary: K-ary Search Tree, Schlegel et at., Damon 2009
- FAST: Fast Architecture Sensitive Tree, Kim et al., SIGMOD 2010
- HT: Chained Hash Table

Index: ART: Performance: Insert

- 16M entries, 4 byte keys



Boolean Expressions

Bxp: Outline

Bxp: Preliminaries

Presentation restricted to column stores.
Algebraic operators needed:

- relation scan: $\texttt{scan}(R)$
- select: σ
- map: χ

Bxp: Preliminaries: scan

- the scan of a relation R is denoted by $\texttt{scan}(R)$.
- it produces RIDs or column indices or pointers into columns

Important: the scan does not include access to columns/attributes. Since this is a costly memory access, it has to be modelled explicitly.

Bxp: Preliminaries: map

The map operator adds a new attribute to a set/bag of input tuples:

$$\begin{aligned} \chi_{A:e'}(e) &:= \{t \circ [A : v] \mid t \in e, v = e'(t)\} \\ \chi_{A_1:e_1,\ldots,A_k:e_k} &:= \chi_{A_k:e_k}(\ldots(\chi_{A_1:e_1}(e))\ldots) \end{aligned}$$

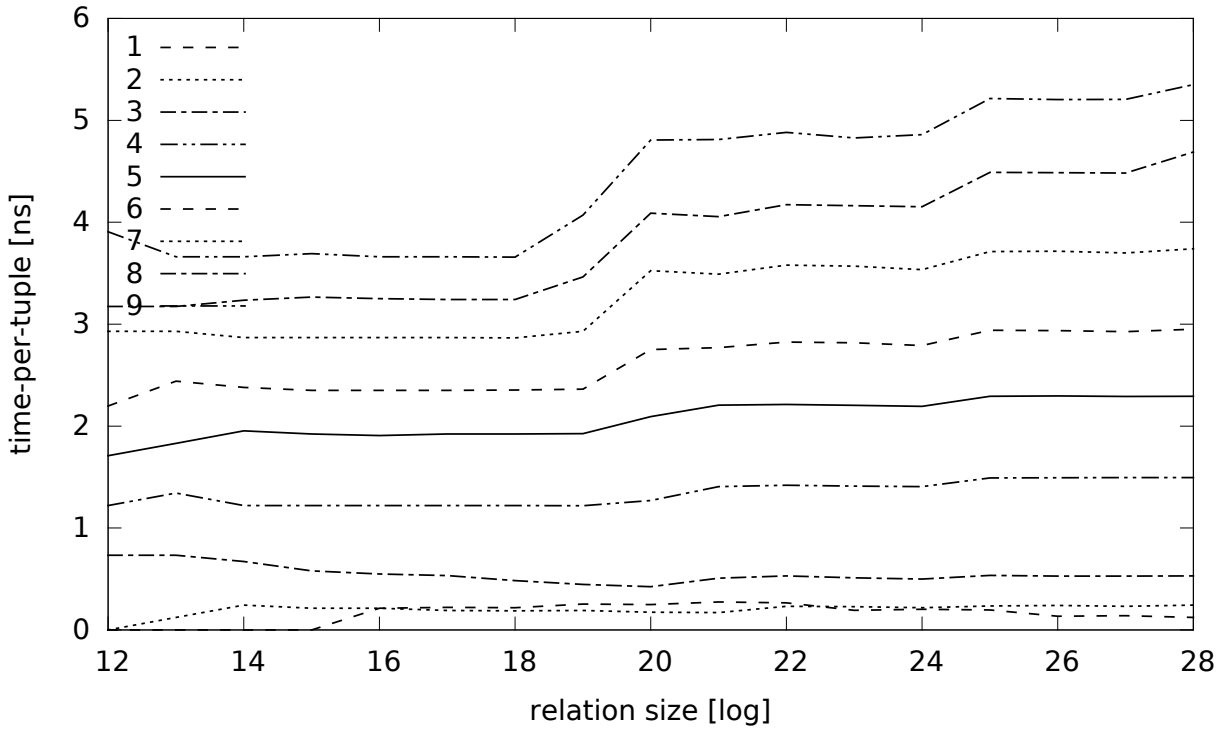
where e is an algebraic expression, A is an attribute name and e_i and e' are expressions. Special case is attribute access, i.e., TID or column pointer dereference:

$$\chi_{*(A_1,\ldots,A_k)}(e)$$

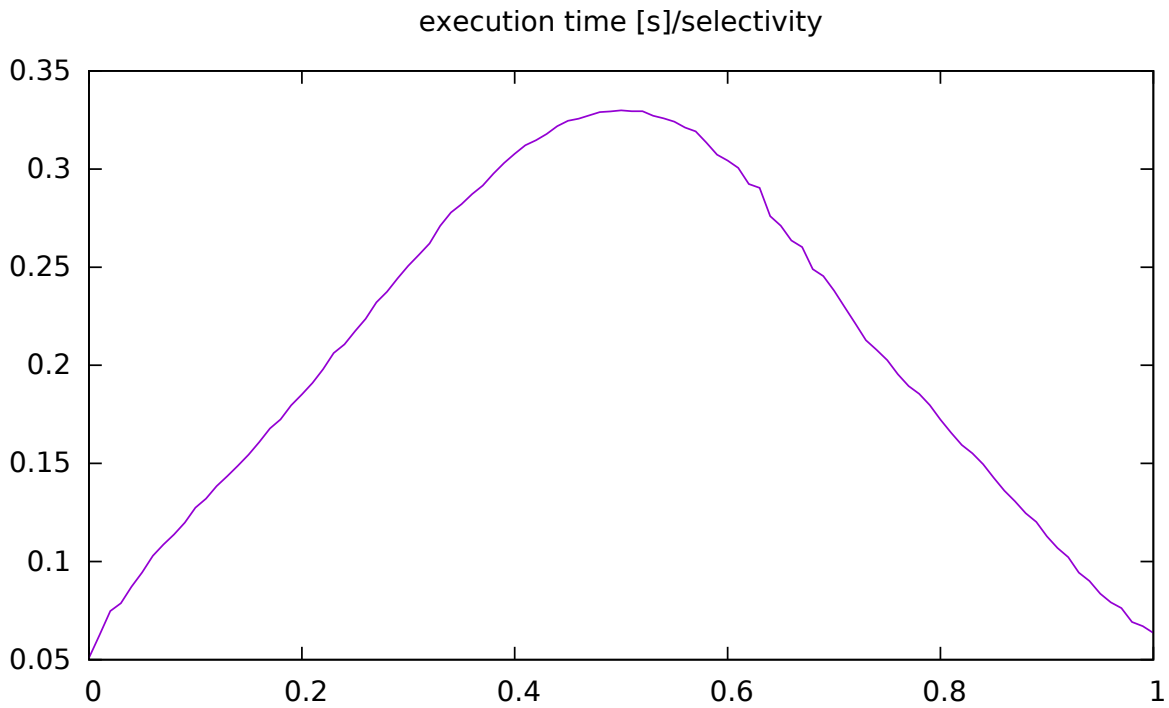
Bxp: Cost Functions

- measured costs
- cost function parameters/notation
- cost functions themselves

Bxp: Measured Column Access Costs in System Tx



Bxp: Measured Selection Costs



Bxp: Measured Costs: Observations

- measurements not absolutely precise (even indeterministic)
- difficult to approximate

Nonetheless, (averaged) measurements are taken to be the truth. Approximation will yield some error. This implies the following question:

- Which error metrics should we minimize?

Bxp: Q-Error

Let x be a value and $\hat{x}$ be an estimate for x. Then, the *q-error* of the estimate $\hat{x}$ is defined as

$$\text{q-error}(\hat{x}) := ||\hat{x}/x||_Q$$

where

$$||y||_Q := \max\{y, 1/y\}$$

Why $||\cdot||_Q$ and not $\|\cdot\|_1$ or $\|\cdot\|_2$ or $\|\cdot\|_\infty$?

Bxp: Q-Error

For an expression e:

- let $\mathcal{C}(e)$ denote the result of some cost function
- let $\mathcal{M}(e)$ denote some measured costs
- let $\mathcal{E} = \{e_1, \ldots, e_k\}$ be a set of plans
- let e_{opt} be the optimal plan for a query Q minimizing $\mathcal{M}(e)$
- let e_{best} be the optimal plan for a query Q minimizing $\mathcal{C}(e)$

We are now interested in the factor by which the true costs of e_{best} are larger than the true costs of the optimal plan e_{opt}.

Bxp: Q-Error: Theorem

If for all $e_i \in \mathcal{E}$

$$||\mathcal{C}(e_i)/\mathcal{M}(e_i)||_Q \leq q$$

for some q, then

$$||\mathcal{M}(e_{best})/\mathcal{M}(e_{opt})||_Q \leq q^2$$

Bxp: Q-Error: Corollary

If for all $e_i \in \mathcal{E}$

$$||\mathcal{C}(e_i)/\mathcal{M}(e_i)||_Q \leq q$$

for some q and for all $e_i \neq e_{\text{opt}}$

$$q < \sqrt{||\mathcal{M}(e_i)/\mathcal{M}(e_{\text{opt}})||_Q},$$

then

$$\mathcal{M}(e_{\text{best}}) = \mathcal{M}(e_{\text{opt}}).$$

Bxp: Q-Error: Proof of Theorem

Since under the cost function $\mathcal{C}$ the plan e_{best} is minimal, we must have

$$\mathcal{C}(e_{\text{best}}) \leq \mathcal{C}(e_{\text{opt}}),$$

and since under $\mathcal{M}$ the plan e_{opt} is minimal, we have

$$\mathcal{M}(e_{\text{opt}}) \leq \mathcal{M}(e_{\text{best}}).$$

Since for all plans e we have $||\mathcal{M}(e)/\mathcal{C}(e)||_Q \leq q$, we can conclude that[1]

$$\begin{aligned} \mathcal{M}(e_{\text{best}}) &\leq q\mathcal{C}(e_{\text{best}}) \\ \mathcal{M}(e_{\text{opt}}) &\geq (1/q)\mathcal{C}(e_{\text{opt}}). \end{aligned}$$

Using all these inequalities, we can derive

$$\begin{aligned} ||\mathcal{M}(e_{\text{best}})/\mathcal{M}(e_{\text{opt}})||_Q &\leq \frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})} \\ &\leq \frac{q\mathcal{C}(e_{\text{best}})}{(1/q)\mathcal{C}(e_{\text{opt}})} \\ &\leq \frac{q\mathcal{C}(e_{\text{opt}})}{(1/q)\mathcal{C}(e_{\text{opt}})} \\ &\leq q^2 \end{aligned}$$

□

[1] $\forall x > 0\ ||x||_Q \leq q \Longrightarrow 1/q \leq x \leq q$

Bxp: Q-Error: Proof of Corollary

Assume $\mathcal{M}(e_{\text{best}}) \neq \mathcal{M}(e_{\text{opt}})$.
Then, by our Theorem we have the following contradiction:

$$\frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})} \leq q^2 < \frac{\mathcal{M}(e_{\text{best}})}{\mathcal{M}(e_{\text{opt}})}$$

□

Bxp: Q-Error: Consequences

As a consequence of the theorem and its corollarly

- it becomes clear that we must minimize the q-error
- we must approximate the measurements such that the q-error is minimized

The latter implies that linear regression, which minimizes l_2 is not appropriate.

We use the following parameters for our cost functions

Notation	Description
R	relation
$A_{(i)}, B_{(i)}, \ldots$	attributes, with and without index
$\mathcal{A}$	set of attributes
$\chi_{*(\mathcal{A})}$	map operator accessing $\mathcal{A}$
a_χ, b_χ	constants for map operator
$deref(d)$	costs of dereferencing d columns
$p_{(i)}$	predicates
$s_{(i)}$, sel$(p_{(i)})$	selectivities for predicates
P	set of predicates, interpreted conjunctively
sel(P)	selectivity of a set of predicates
e	some algebraic expression (plan)
a_s, b_s	constants for scan operator
a_{in}, a_{out}	constants for processing input/output tuples
$\mathcal{B}(s)$	branch misprediction cost for selectivity s
$\mathcal{C}(e)$	cost function applied to e, estimated runtime

Bxp: Cost Model

$$
\begin{aligned}
\mathcal{C}(scan(R)) &= |R| * a_s + b_s \\
\mathcal{C}(\chi_{*(\mathcal{A})}(e)) &= |e| * (deref(1,n) + a_\chi) + b_\chi \\
\mathcal{C}(p_1 \& p_2) &= \mathcal{C}(p_1) + \mathcal{C}(p_2) + \mathcal{C}(\&) \\
\mathcal{C}(p_1 \&\& p_2) &= \mathcal{C}(p_1) + \mathcal{B}(s_1) + s_1 \mathcal{C}(p2) \\
\mathcal{C}(\sigma_p(e)) &= |e| * (\mathcal{C}(p) + \mathcal{B}(sel(p)) + a_{in} + sel(p) * a_{out})
\end{aligned}
$$

Observe:

- cost functions mostly linear with some non-linear components like $\mathcal{B}$
- cost functions contain constants: calibration is needed
- selectivities/cardinalities must be known

Bxp: Cardinality Estimation

Let $P = \{p_1, \ldots, p_z\}$ be the set of predicate used in some conjunctive query.
Then, we need

$$\text{sel}(\bigwedge_{i \in S} p_i)$$

for all $S \subseteq \{1, \ldots, z\}$.
We discuss only one possibility to derive these: gamma-sampling.

Bxp: Cardinality Estimation: gamma-sampling (1)

Let $P = \{p_1, \ldots, p_z\}$ denote a set of z predicates. For a subset of predicates $P' \subseteq P$, we denote by $\beta(P')$ the formulae

$$F_\beta(P') = \bigwedge_{p_i \in P'} p_i,$$

and by $\gamma(P')$ the formulae

$$F_\gamma(P') = \bigwedge_{p_i \in P'} p_i \wedge \bigwedge_{p_i \notin P'} \neg p_i.$$

(F_β are conjuncts of predicates and F_γ are minterms.)

Bxp: Cardinality Estimation: gamma-sampling (2)

The selectivities of these predicates are denoted by

$$\beta(P')$$

and

$$\gamma(P')$$

For our algorithm, we need the vector β, which gathers the $\beta(P')$ for all P'. The procedure `getGamma` presented below will give us γ. Hence, we need a method to convert γ to β.

Bxp: Cardinality Estimation: gamma-sampling (3)

A technicality:

- every subset $P' \subseteq P$ can be expressed as bitvector $\mathrm{bv}(P')$ of length $|P|$
- $\mathrm{bv}(P')$ can be interpreted as a positive integer whose representation it is

Subsequently, we identify these two different interpretations of the same bitpattern.

Bxp: Cardinality Estimation: gamma-sampling (4)

Let $n = 2^z$. Define the *complete design matrix* $C \in \mathbb{R}^{n,n}$ as

$$C(i,j) = \begin{cases} 1 & \text{if } j \supseteq i \\ 0 & \text{else} \end{cases}$$

where $j \supseteq i$ denotes the fact that every bit set to one in i is also set in j, i.e., $i = i\&j$ and i, j range from 0 to $2^z - 1$.
C is binary, non-singular, upper triangular, and persymmetric.

Bxp: Cardinality Estimation: gamma-sampling (5)

The complete design matrix C allows us to go from γ to β by

$$C\gamma = \beta$$

Bxp: Cardinality Estimation: gamma-sampling (6)

```
getGamma(p, z, S)
    // p is vector of predicates,
    // z its length,
    // S is the sample
    int n = (1 << z);
    // array of counters initialized to zero
    int c_gamma[n] = 0;
    // for all sample tuples in S
    for(s : S)
        int k = 0; // accumulated results for predicate evaluations
        for(int i = 0; i < z; ++i) // for each predicate
            // p[i](s): evaluate pi on sample tuple s
            k |= (p[i](s) << i);
        ++c_gamma[k];
    return c_gamma/|S|; // componentwise division
```

Bxp: Conjunctive Queries

- sort by increasing selectivity s
  ignores different costs, relies on independence assumption (IA)
- sort by increasing rank ($r = \frac{s-1}{c}$)

s selectivity of some predicate, c cost of some predicate.
Ad 2:

- selectivity 'changes' if IA does not hold
- BMP costs depend on selectivity, thus cost change
- costs of a predicate might change if common subexpressions occur
- selectivity of a predicate changes badly in case of implications

Bxp: IA Example I

assume some attribute A contains uniformly randomly distributed numbers in $[1, 100]$
then

$$\begin{aligned} \text{sel}(A \leq 51) &= 0.51 \\ \text{sel}(A \geq 50) &= 0.51 \end{aligned}$$

However, after $\sigma_{A \leq 51}$ has been applied,

$$\text{sel}(A \geq 50) = 0.02$$

Bxp: IA Example II

Two simple predicates on a CAR relation from the DMV:

- make = 'HONDA'
- model = 'ACCORD'

Bxp: conjunctive queries: $\texttt{DP}_{\texttt{sel}}$

- no independence assumption (IA)
- branch misprediction (BMP) costs
- common subexpression elimination (CSE)
- build plans using both, & and &&
- uses dynamic programming

Bxp: conjunctive queries: $\texttt{DP}_{\texttt{sel}}$: BuildPlans

BUILDPLANS(p, e)

Input: a selection predicate p
an expression e (partial plan)
Output: plan container B

1 $X_e = \cup_{p_i \in e} X_{p_i}$
2 $X_{p|e} = X_p \setminus X_e$ // outstanding maps
3 $B = \{\sigma_p(X_{p|e}(e))\}$
4 **if** $e ==$ $\sigma_{p'}(X_{p|e}(e'))$
5 $B += \sigma_{p' \& p}(X_{p|e}(e'))$
6 $B += \sigma_{p' \&\& p}(X_{p|e}(e'))$
7 **return** B

Bxp: conjunctive queries: $\texttt{DP}_{\texttt{sel}}$: DpInsert

DPINSERT(e, P, DP)

Input: an expression e
a set of predicate(s) P
a DP table

Output: none, affects DP

1 **if** $DP[P]$ == null $\vee\ \mathcal{C}(DP[P]) > \mathcal{C}(e)$
2 $\quad DP[P] = e$

Bxp: conjunctive queries: $\texttt{DP}_{\texttt{sel}}$

DPSEL

Input: a set $P = \{p_0, \ldots, p_{n-1}\}$ of predicates
Output: an optimal plan

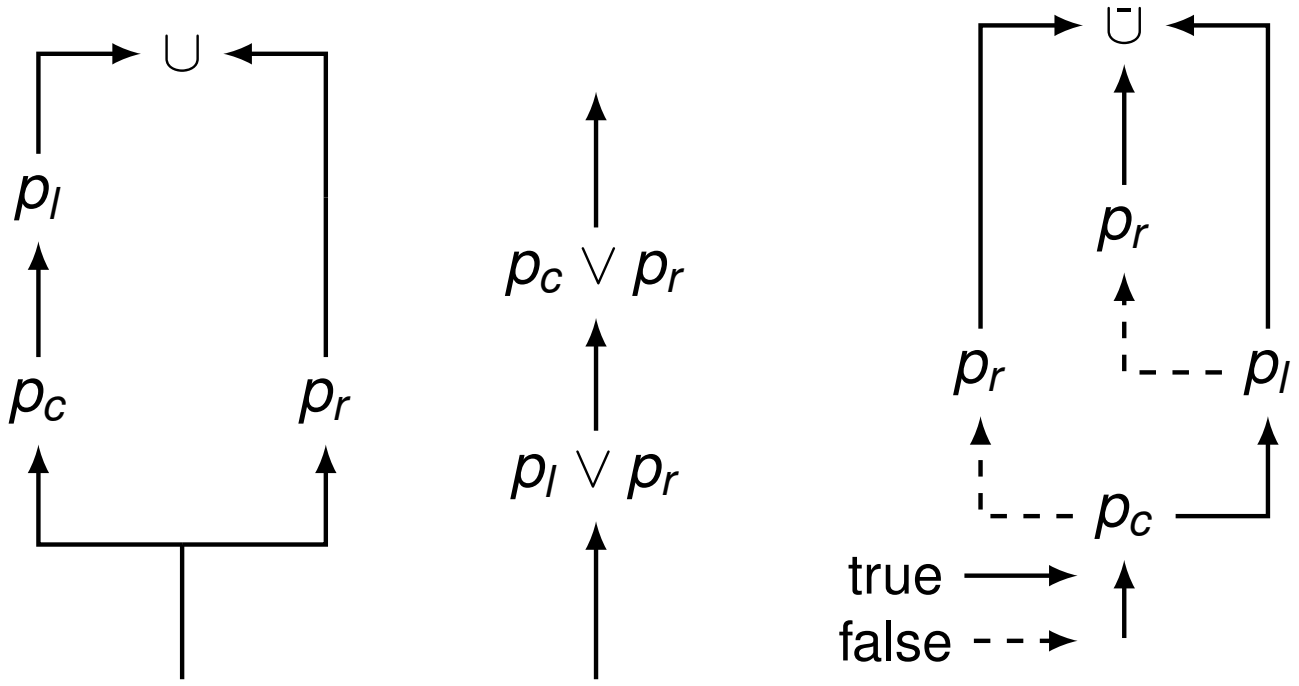
1 $DP =$ an empty DP table, size $\rightarrow 2^n$
2 $DP[\varnothing] = scan(R)$
3 **for each** $0 \leq i < 2^n - 1$ **ascending**
4 $\quad P' = \{p_k \in P \mid (\lfloor i/2^k \rfloor \bmod 2) = 1\}$
5 $\quad$ **for each** $p_j \in P \setminus P'$
6 $\quad\quad$ **for each** $e_j \in$ BUILDPLANS$(p_j, DP[P'])$
7 $\quad\quad\quad$ DPINSERT$(e_j, P' \cup \{p_j\}, DP)$
8 **return** $DP[P]$

Bxp: disjunctions: plan alternatives (1)

- disjunctive normal form (DNF) plans require duplicate eliminating union
- conjunctive normal form (CNF) plans typically imply redundant evaluations
- bypass plans are the best known choice

main idea bypass plans: select operator gets two output streams: one for tuples satisfying the selection predicate, one for those that do not. (union becomes simple union [no dup elim])

Bxp: disjunctions: plan alternatives (2)



(a) DNF plan (b) CNF plan (c) Bypass plan

Figure: Evaluation plans for the query $(p_c \wedge p_l) \vee p_r$

Bxp: disjunction: plan alternatives (3)

- neither CNF nor DNF plans are opimal
- both require normalization which may lead to exponential blow up
- thus: we are left with bypass plans

Question:

How to generate optimal bypass plans?

Bxp: disjunction: prerequisites

- an *assignment* is a set of elements of the form $p_i \leftarrow v$ where v is a truth value
- let b be a boolean expression and A be an assignment then $b[A]$ denotes the replacement of the predicates in A by their assigned truth values and subsequent simplification

example: for $A = \{p_2 \leftarrow \text{false}\}$:

$$\begin{aligned}(p_1 \wedge p_2) \vee (p_3 \wedge p_4)[A] &\equiv (p_1 \wedge \text{false}) \vee (p_3 \wedge p_4) \\ &\equiv (p_3 \wedge p_4)\end{aligned}$$

Note: the same result occurs for $A = \{p_1 \leftarrow \text{false}\}$.

Bxp: disjunction: prerequisites

Consequences:

- Remember: the same result occurs for $A = \{p_1 \leftarrow \text{false}\}$.
- However: selectivities for p_3 and p_4 may differ!
- Thus: cannot use boolean expressions to index Memo table!
- Hence: we use assignments to index Memo table

We use top-down plan generation with memoization. This is why the DP table is renamed to Memo table.

Bxp: disjunction: $\texttt{TD}_{\texttt{byp}}$

TDBYP(*e*, *Bxp*, *Asg*, *branch*)

```
    // Input: partial plan e
                a Boolean expression Bxp
                an assignment Asg
                flag branch
    // Output: best plan
1   if Memo[Asg]
2         return Mem[Asg]
3   bestcost = ∞
4   bestplan = NULL
```

```
1   for each p ∈ {getPredicates(Bxp)}
2         e′ = BUILDPLANS(p, e, branch)
3         A = {p ← TRUE}
4         e+ = TDSIM(e′, Bxp[A], Asg ∪ A, TRUE)
5         A = {p ← FALSE}
6         e− = TDSIM(e′, Bxp[A], Asg ∪ A, FALSE)
7         cost = Cost(e+) + Cost(e−) + Cost(e′)
8         if bestplan == NULL or bestcost > cost
9               bestplan = [e′, e+, e−]
10              bestcost = cost
11  Memo[Asg] = bestplan
12  return bestplan
```

Bxp: disjunction: BuildPlans

BUILDPLANS(p, e, *branch*)

// **Input:** a selection predicate p
a partial plan e
flag *branch*
// **Output:** (partial) plan

1 $X_e = \cup_{p_j \in e} X_{p_j}$
2 $X_{p|e} = X_p \setminus X_e$ // outstanding maps
3 **if** e == *scan*(R)
4 **return** $\sigma_p(\chi_{*P}(e))$
5 **elseif** *branch* == TRUE
6 // at this point we know that e is not a scan
7 $e = \sigma_p(X_{p|e}(\sigma^+_{p_j}(e')))$
8 **else**
9 $e = \sigma_p(X_{p|e}(\sigma^-_{p_j}(e)))$
10 **return** e

Bxp: disjunction: $\texttt{TD}_{\texttt{acb}}$

prune search space while preserving optimality:

- branch-and-bound pruning
- here specialization: accumulated cost bounding
- prune execution of cost exceeds a given budget
- problem: reoptimization for rising budgets
- standard solution: exponential budget growth
- Memo[Asg].LB returns 0 by default
  (lower bound for best plan for Asg)
- initial call: budget $b = \infty$ (or heuristic like BDC)

Bxp: disjunction: $\texttt{TD}_{\texttt{acb}}$

TDACB(e, *Bxp*, *Asg*, *branch*, b)

*// **Input:*** partial plan e, a Boolean expression *Bxp*,
an assignment *Asg*, flag *branch*, cost budget b
*// **Output:*** best plan

1 **if** $Memo[Asg] \neq$ NULL and $Cost(Memo[Asg]) \leq b$
2 **return** $Memo[Asg]$
3 **if** $Memo[Asg].LB \geq b$
4 **return** NULL
5 **if** $Memo[Asg].LB > 0$
6 $b = \text{MAX}(b, Memo[Asg].LB * 2)$
7 $bestcost = \infty$
8 $bestplan =$ NULL

```
1   for each p ∈ {getPredicates(Bxp)}
2         e′ = BUILDPLANS(p, e, branch)
3         b′ = MIN(b, bestcost) − Cost(e′)
4         A = {p ← TRUE}
5         e+ = TD_ACB(e′, Bxp[A], Asg ∪ A, TRUE, b′)
6         if e+ ≠ NULL
7               b′ = b′ − Cost(e+)
8               A = p ← FALSE
9               e− = TD_ACB(e′, Bxp[A], Asg ∪ A, FALSE, b′)
10              if e− ≠ NULL
11                    cost = Cost(e+) + Cost(e−) + Cost(e′)
12                    if bestplan == NULL or bestcost > cost
13                          bestplan = [e′, e+, e−]
14                          bestcost = cost
```

```
1  // If no valid plan was found with budget b
2  if bestplan.e+ == NULL or bestplan.e− == NULL
3        Memo[Asg].LB = b
4        return NULL
5  Memo[Asg] = bestplan
6  return Memo[Asg]
```

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 book by Cormode, Garofalakis, Hass, Jermaine. The latter contains an overview of many different estmation techniques.

Sketches for Join Size Estimation

1. Tug-Of-War (AGMS sketch)
2. FastAGMS sketch

Frequency Moments

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

The Random Variables

Let $\zeta_i \in \{-1, +1\}$ be a random variable. Then its expected value is $E(\zeta_i) = 0$. Define a random variable

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

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

$$X = Z^2.$$

We show that for two-way independent ζ_i we have

$$E(X) = F_2$$

Proof

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

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

□

Variance

Next, we show that

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

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

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

(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

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

□

AGMS Sketch for Self-Join Size

- variance bounded but pretty high
- use median of averages to decrease variance

We need:

- Let s_1, s_2 be positive integers.
- Let $\zeta_{i,j}$ be 4-universal hash functions.

AGMS Sketch for Self-Join Size

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.

AGMS Sketch for Self-Join Size

Increasing s_1 increases precision; increasing s_2 increases confidence:

Theorem

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

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

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

AGMS Sketch for Join Size

Given relations R_1 and R_2. The counters Z^1 and Z^2 are defined as

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

$$Z^2 := \sum_{i=1}^{n} \zeta_i g_i$$

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

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

AGMS Sketch for Join Size

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.

AGMS Sketch: Code for insert

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*.

AGMS Sketch: Code for producing the estimate

```
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();
    // 2. 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];
```

AGMS Sketch: Discussion

- relatively precise
- insertion time proportional to number of counters

FastAGMS Sketch: Overview

- use s_2 sketch vectors Z_i $(1 \leq i \leq s_2)$ of length s_1
- instead of updating $s = s_1 s_2$ counters, only s_2 counters are updated.
- use hash function to determine which counter in each sketch vector is updated.

FastAGMS Sketch: Hash Functions

- let $U = \{v_1, \ldots, v_n\}$ be the domain of the join attribute.
- 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.

The hash functions:

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

FastAGMS Sketch: Build

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 $j = 0$; $j < s_2$; ++j)

$Z_{\text{aRelNo}}[j * s_1 + h_{1,j}(v)]$ += aCount * $h_{2,j}(v)$;

FastAGMS Sketch: estimate

```
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];
```

Parallelism

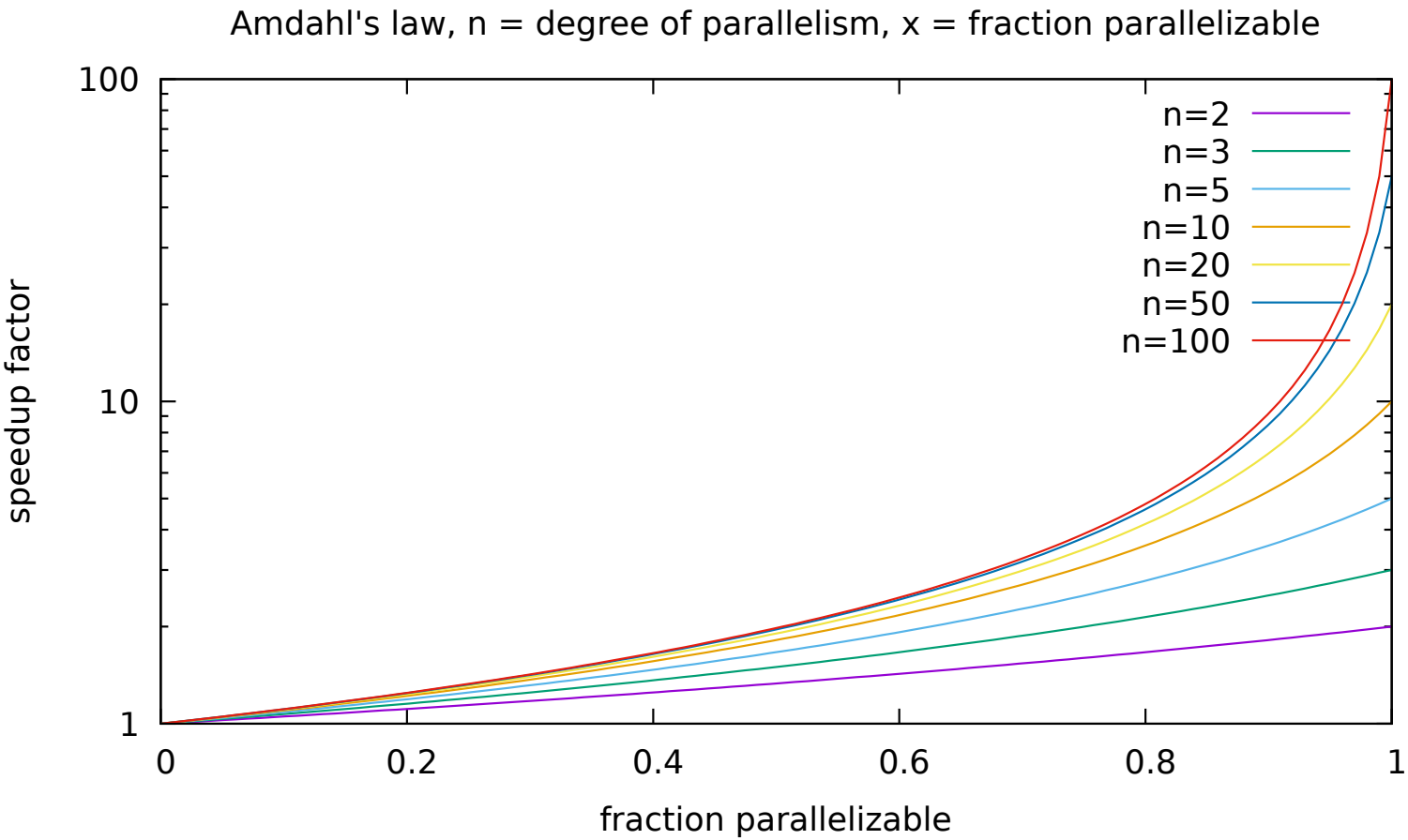
Parallelism: Amdahl's Law

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

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

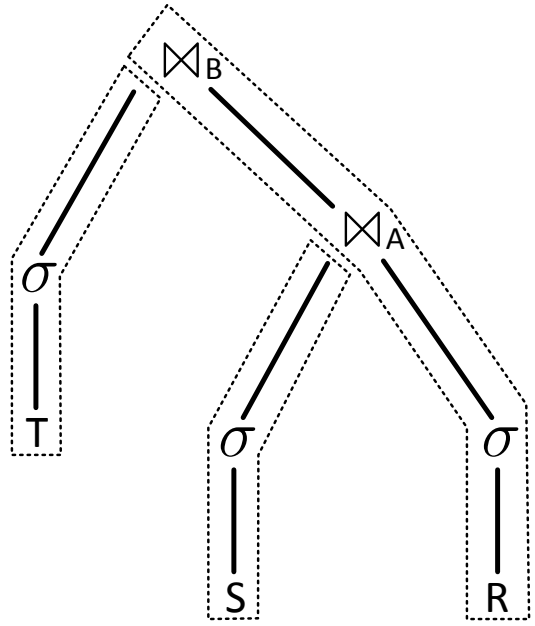
Parallelism: Amdahl's Law

Plotting this formula for different *n* results in:



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

Parallelism: Kinds



kinds of parallelism

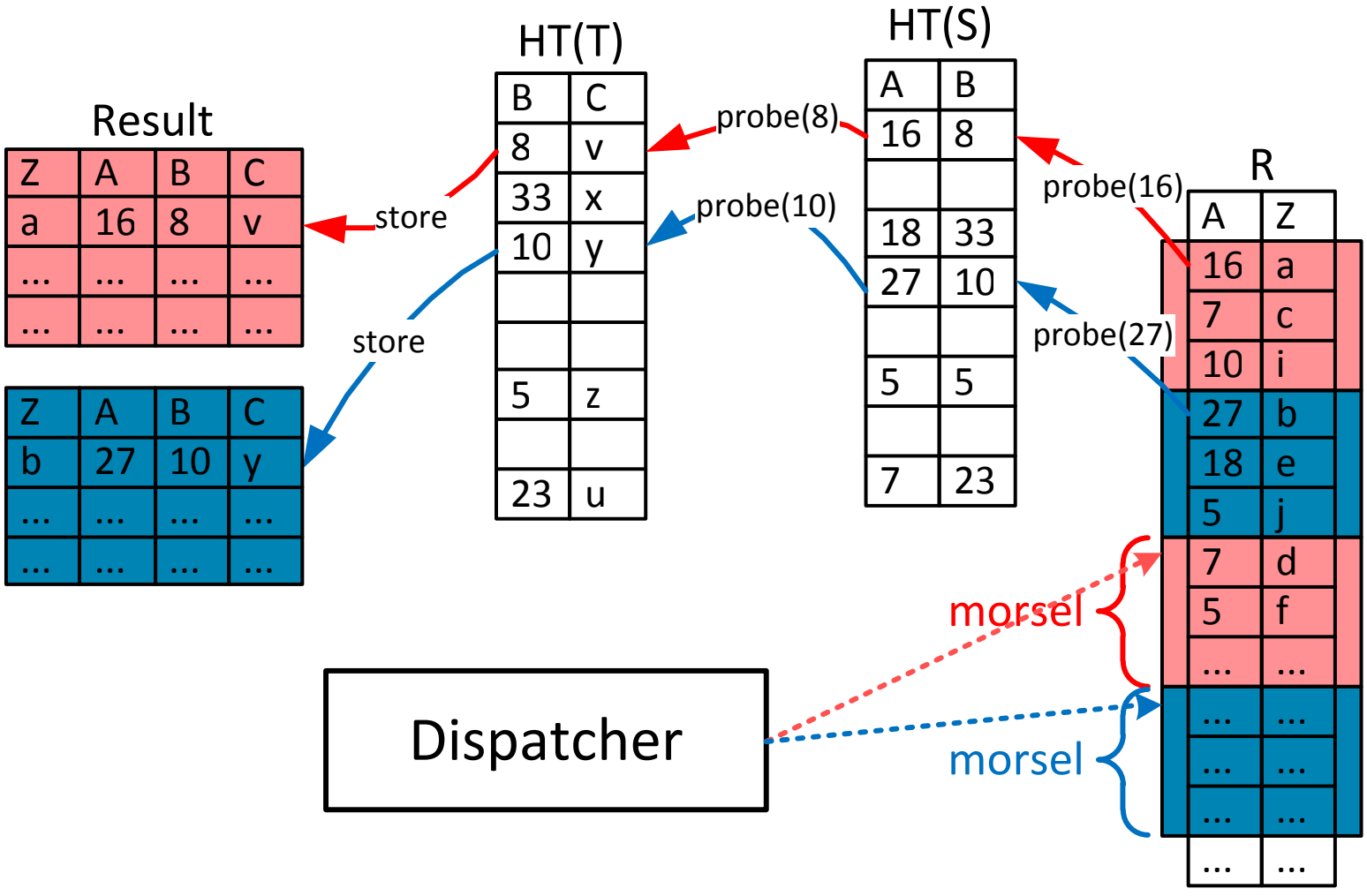
- inter-query parallelism
  - run independent queries in parallel
- intra-query parallelism:
  - partition relation and process partitions in parallel (within strands)
  - process indendent strands in parallel (*bushy parallelism*)

Parallelism: Morsel-Driven

- 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

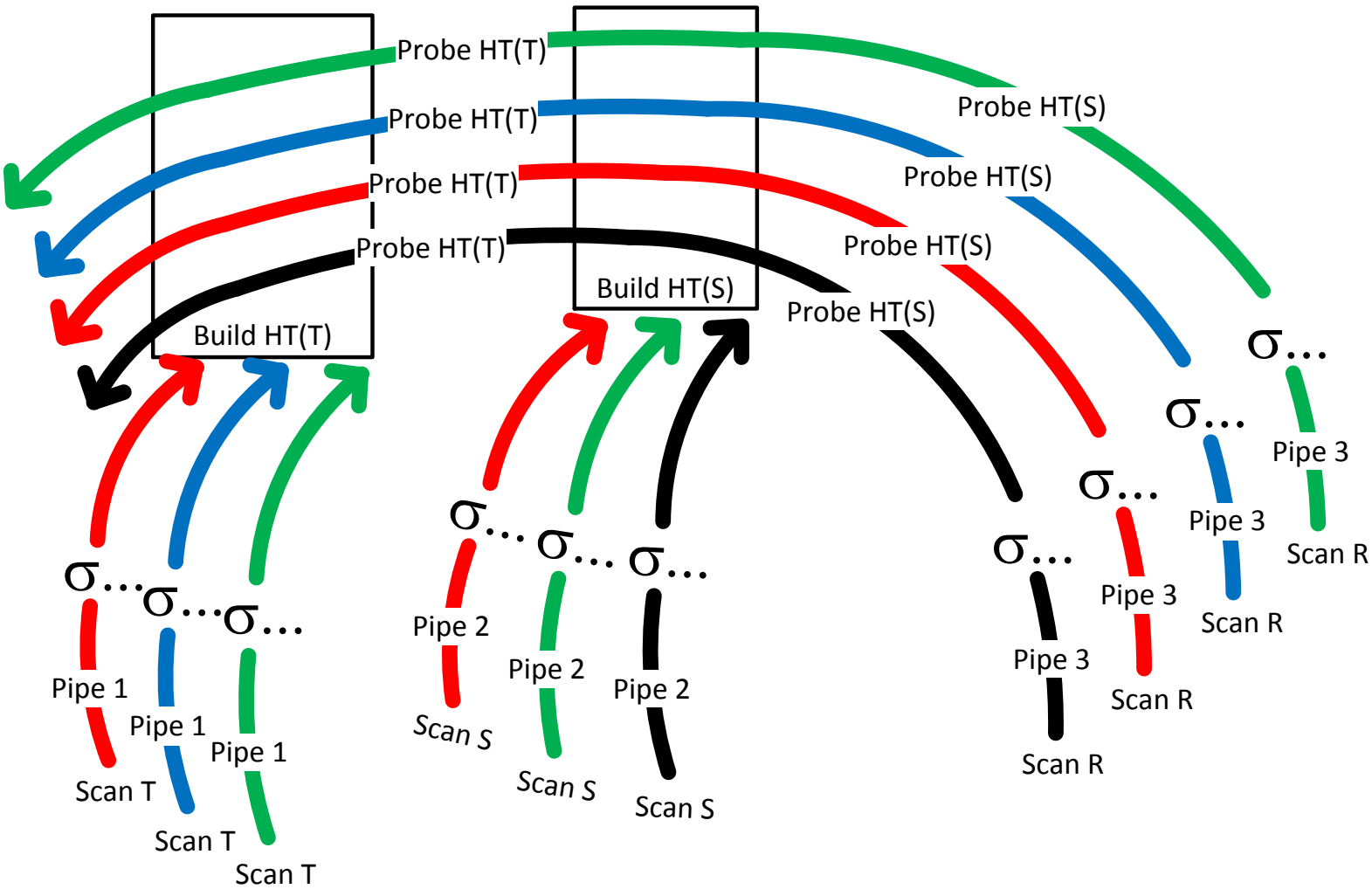
Parallelism: Morsel-Driven

General idea of morsel-driven parallelism for $R \bowtie_A S \bowtie_B T$. The following picture shows the details of the last strand of the above plan:



Parallelism: Morsel-Driven

A complete picture for the whole plan looks as follows:



Parallelism: Morsel-Driven

How do we achieve NUMA-awareness?

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

Parallelism: Morsel-Driven: Join: Build

Build:

- build-phase split into two phases:

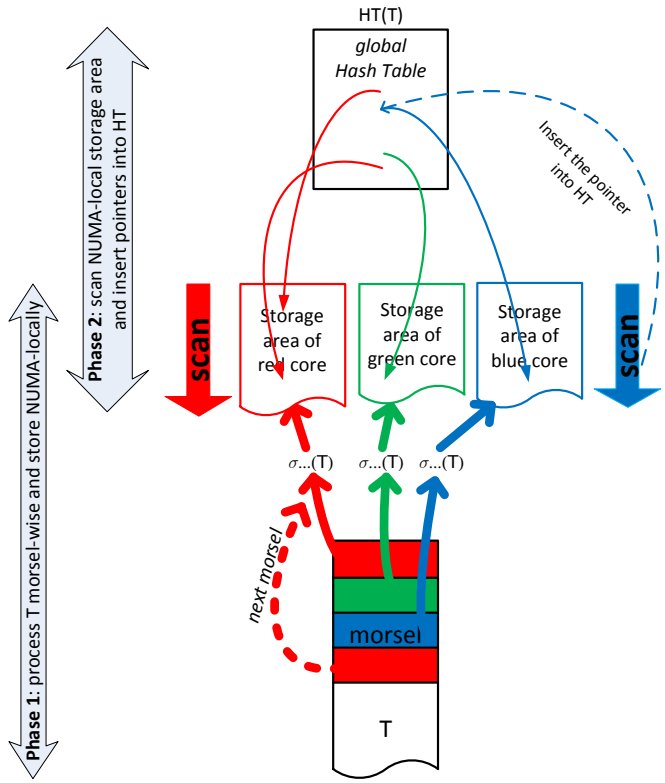
  Mat materializes the input
  HtBuild builds the hash-table

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

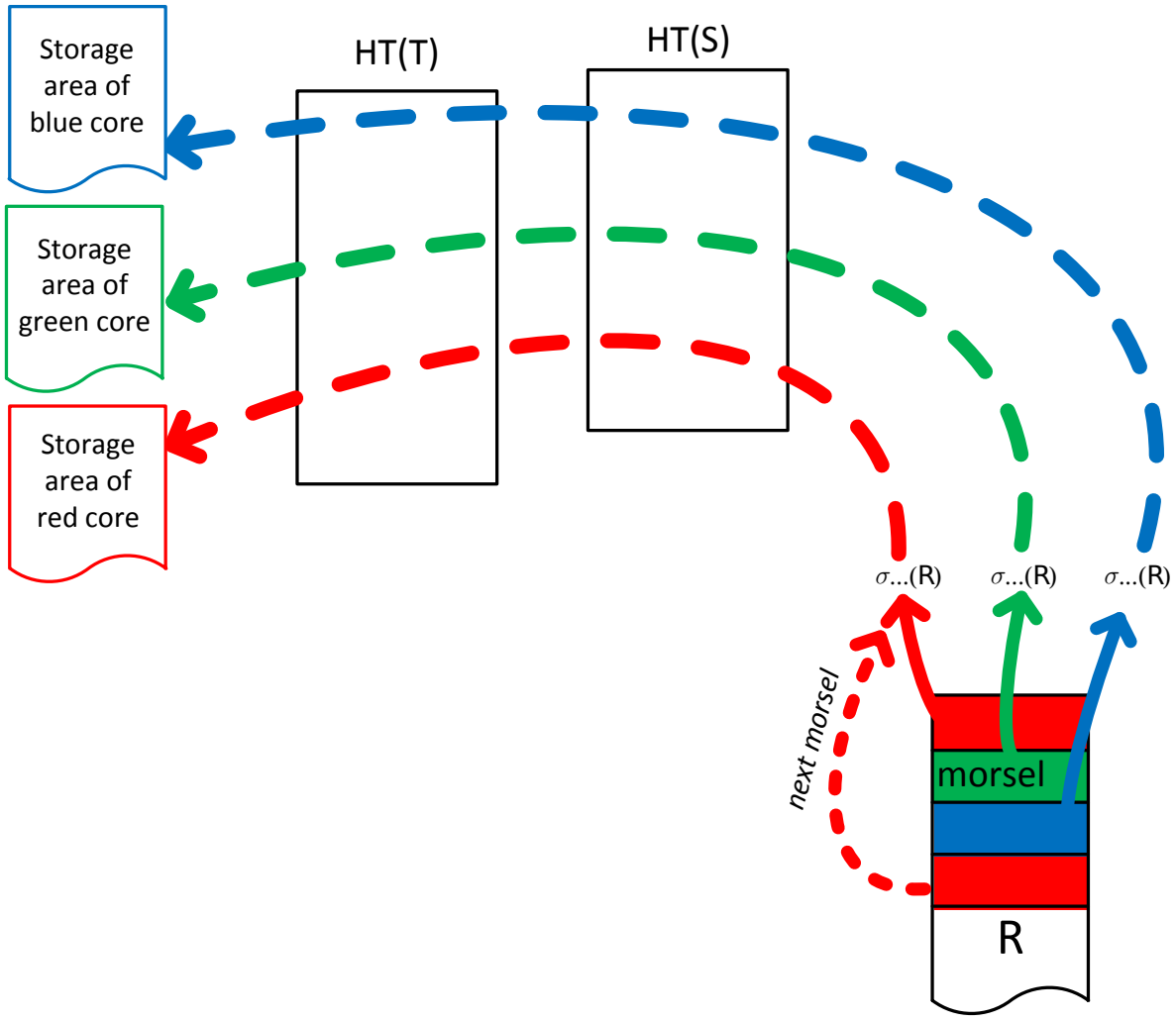
[Note: after materialization the exact size of the input relation is available and it can be used to allocate a hash table of perfect size.]

Parallelism: Morsel-Driven: Join: Build

Colors encode worker threads confined to NUMA-nodes and memory areas belonging to NUMA-nodes.



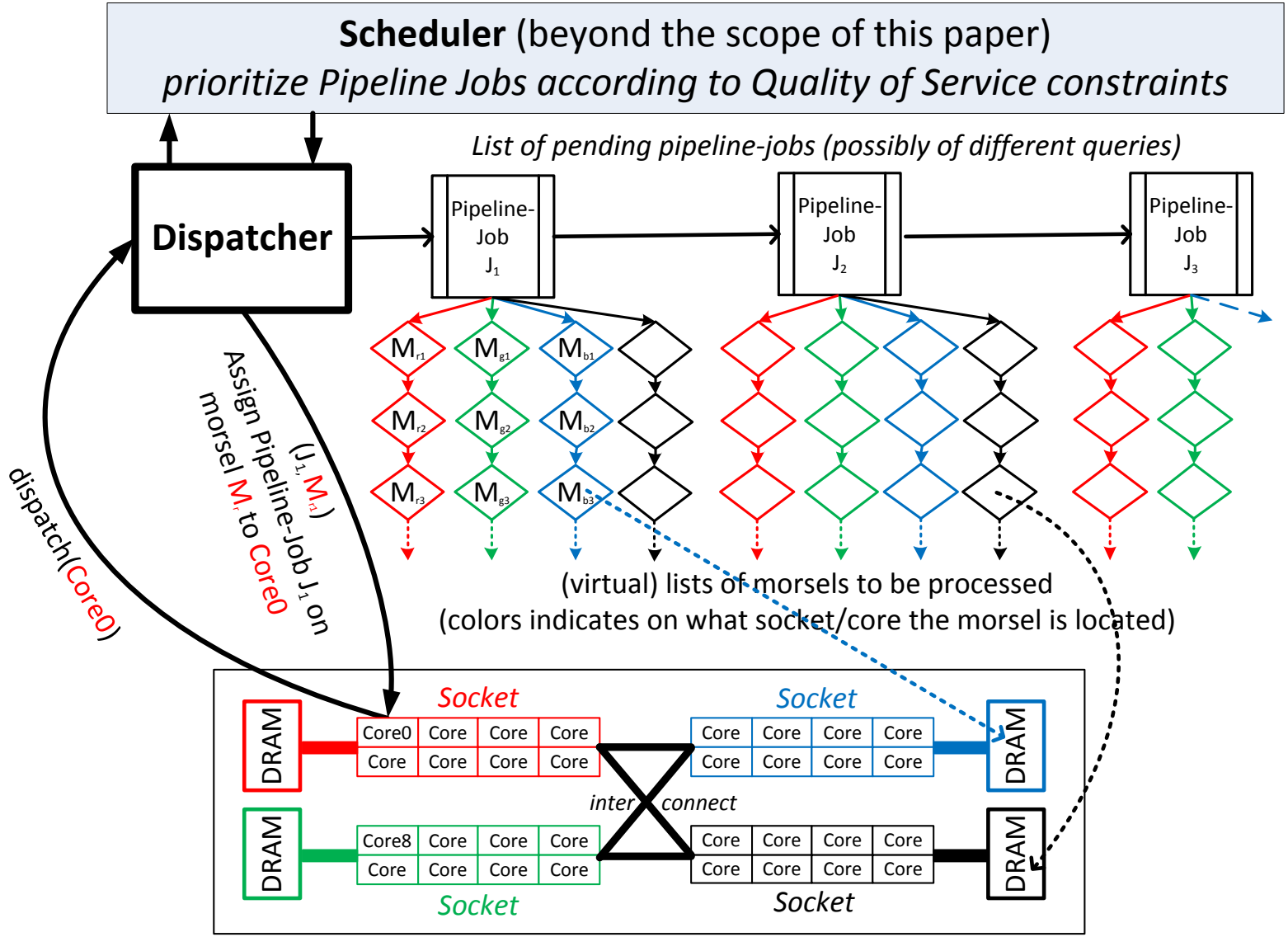
Parallelism: Morsel-Driven: Join: Probe



Parallelism: Morsel-Driven: Join: Details

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

Parallelism: Morsel-Driven: Join: Dispatcher



Example NUMA Multi-Core Server with 4 Sockets and 32 Cores

Parallelism: Morsel-Driven: Join: 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

Transaction Management

Transaction Management

- Lock Manager
- Log Manager

Transaction Management: Updates

Handling updates

- in-place
- delta/staging

Lock Manager

Compabibility matrix for multi-granularity locking:

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.

Lock Manager

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

conversion 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

Lock Manager: Grey/Reuter

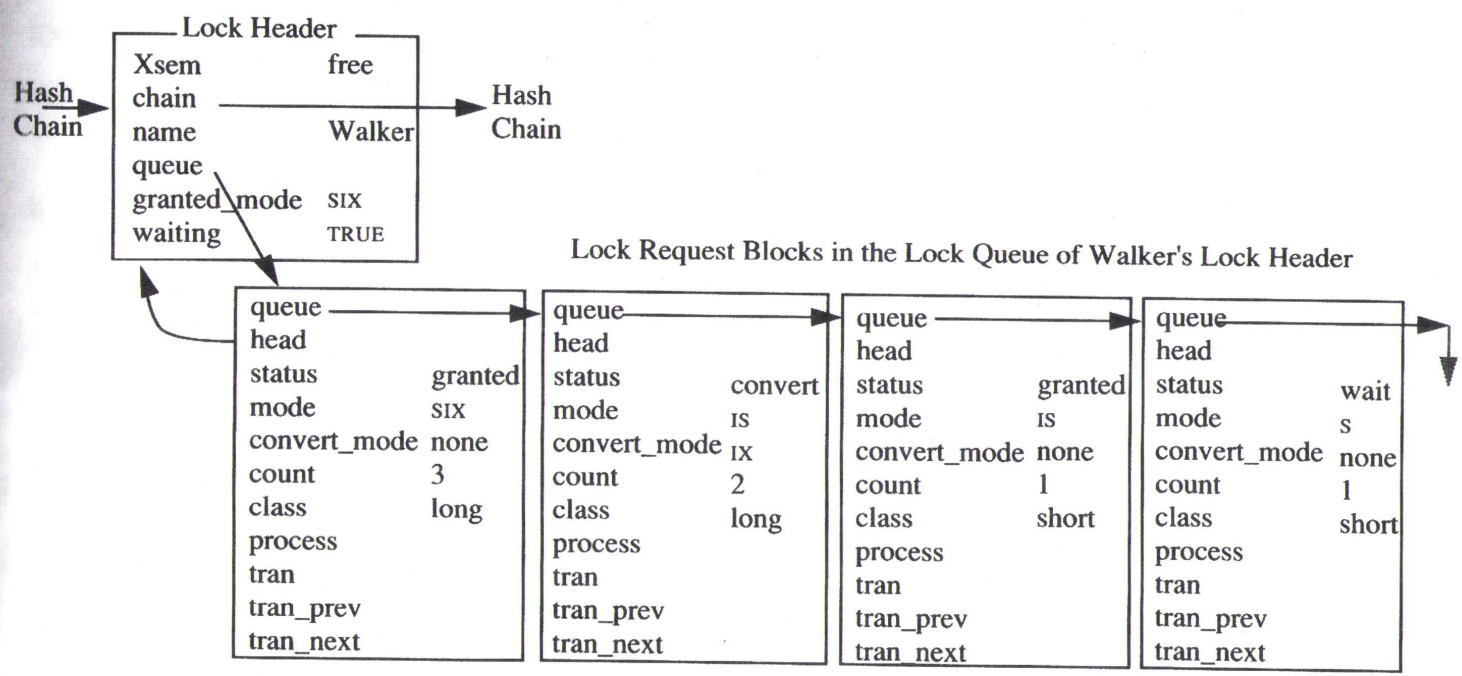


Figure 8.9: Four lock request blocks queued on a lock named Walker. Two requests are granted; one is waiting for a conversion, and one is waiting because the conversion is waiting. To simplify the display, some pointers are not diagrammed. Did the conversion arrive before the waiting S lock request?

Lock Manager: Grey/Reuter: TaCB

Transaction Control Block:

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

Lock Manager: Grey/Reuter: Interface

```
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);
```

Lock Manager: Grey/Reuter: Major Components

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.

Lock Manager: Grey/Reuter: Enum LOCK_MODE

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

Lock Manager: Grey/Reuter: Lock Class

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
};
```

Lock Manager: Grey/Reuter: Hash Table

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

Lock Manager: Grey/Reuter: Lock Head

```
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?
};
```

Lock Manager: Grey/Reuter: Lock Status

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

Lock Manager: Grey/Reuter: Lock Request

```
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 Manager: Grey/Reuter: `lock` (1)

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()
...
```

Lock Manager: Grey/Reuter: `lock` (2)

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
}
```

Lock Manager: Grey/Reuter: `lock` (3)

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)
}
```

Lock Manager: Grey/Reuter: `lock` (4)

Part 4: new lock request by this transaction

```
if(request == NULL) { // new request
    request = lock_request_get(aLock, aMode, aClass); // allocate lock reques
    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);
        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;
```

Lock Manager: Grey/Reuter: `lock`

Remarks:

- sporadic wake-ups
- race conditions (see footnote 5 on page 475 in book by Gray/Reuter
- observe state-machine on lock status
- some systems use bitmap of locks instead of max in `granted_mode`

Lock Manager: Grey/Reuter: `unlock` (1)

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      IRes;           // return code
...
}
```

Lock Manager: Grey/Reuter: `unlock` (2)

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;
}
```

Lock Manager: Grey/Reuter: `unlock` (3)

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;
}
```

Lock Manager: Grey/Reuter: `unlock` (4)

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;
}
```

Lock Manager: Grey/Reuter: `unlock` (5)

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->_mo
    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_
            wakeup(request->_process);
        } else {
            lock->_waiting = true; break; // FIFO
```

Lock Manager: Grey/Reuter: `unlock` (6)

Part 6 does the latch release and return

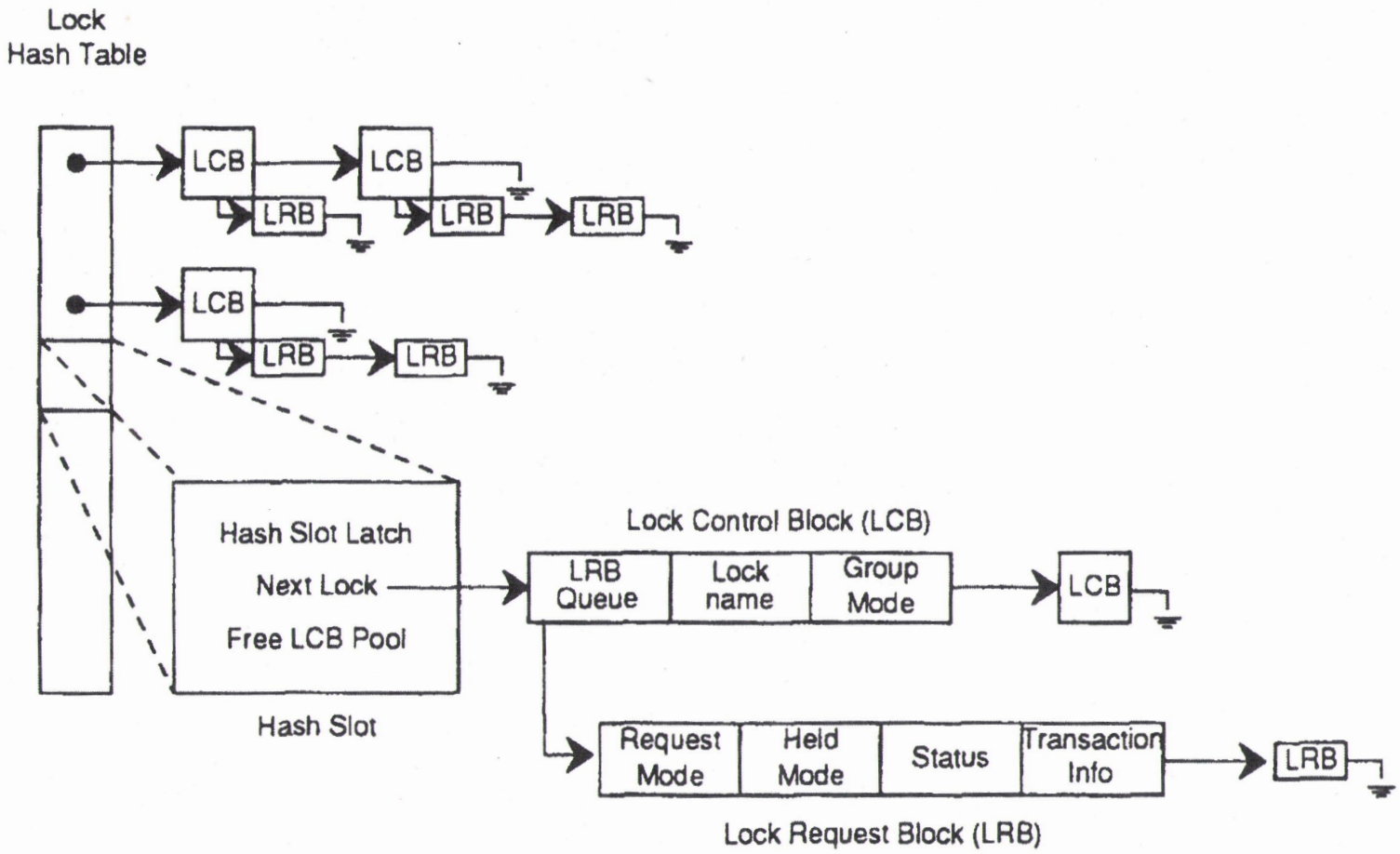
```
    ...
    A: release(lock->_latch);
    B: release(lock_hash[bucket]._latch);
    return LOCK_OK;
}
```

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

Lock Manager: Starburst

- segment: as ususal
- LCB: lock control block
- LRB: lock request block
- note: free LCB pool in slot

Lock Manager: Starburst

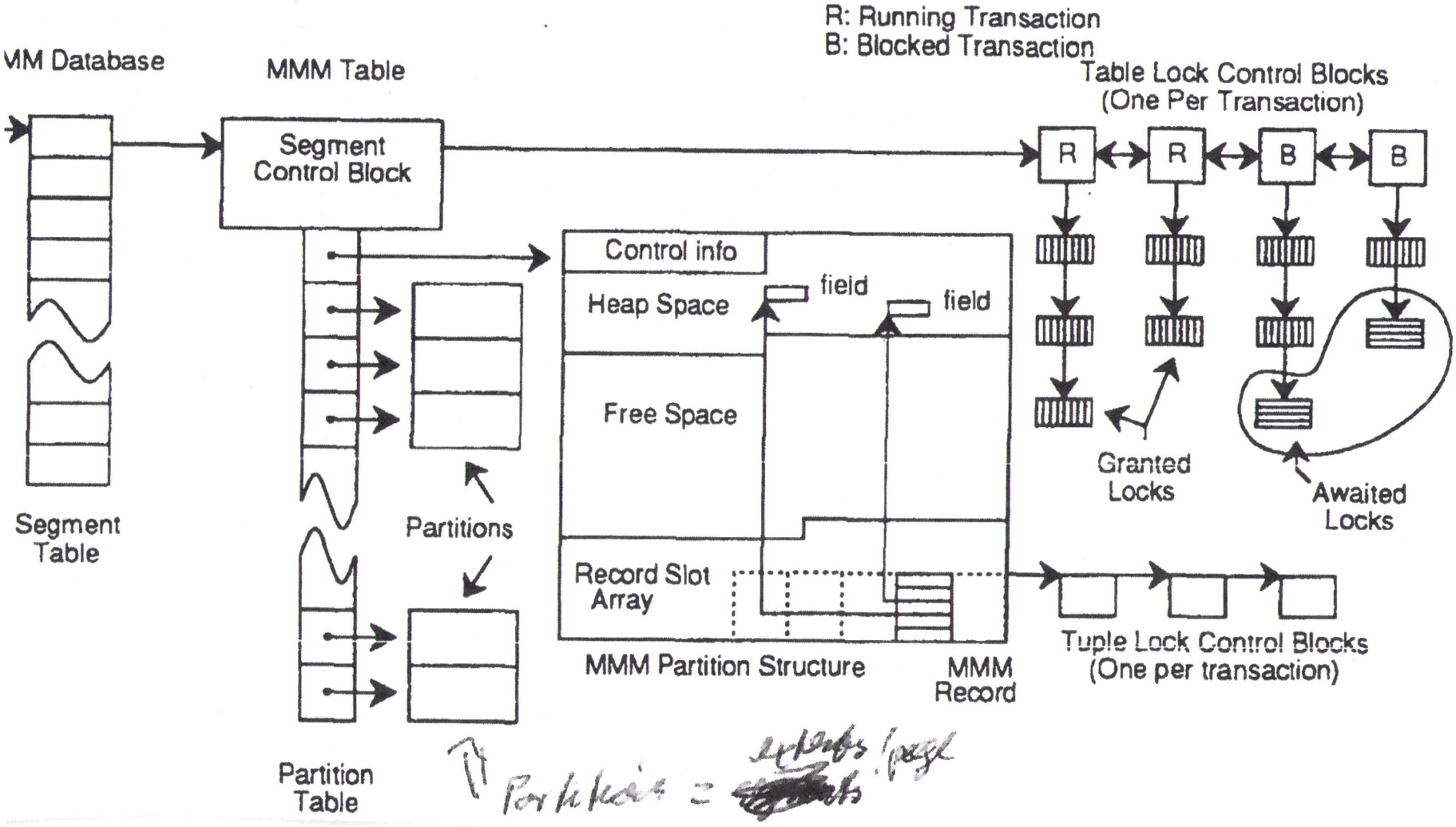


Lock Manager: Starburst MM

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
- MM LM allows for lock escalation and deescalation (dynamically)
- partition: fixed size (similar to page)
  slots contain real main memory pointers to tuples withing partition
- segment: variable number of partitions

Lock Manager: Starburst MM



Lock Manager: Fekete

not this semester

The End