

Data Processing on Modern Hardware

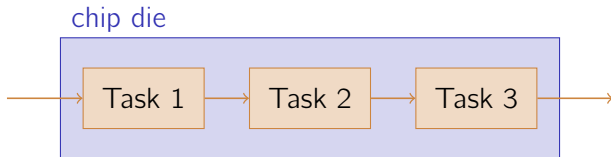
Jens Teubner, TU Dortmund, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

Summer 2014

Part IV

Vectorization

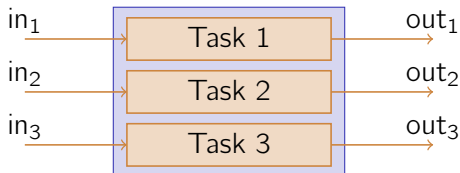
Pipelining is one technique to leverage available **hardware parallelism**.



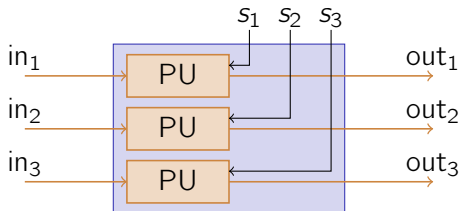
- Separate chip regions for individual tasks execute independently.
- Advantage: Use parallelism, but maintain **sequential execution semantics** at front-end (here: assembly instruction stream).
- We discussed problems around **hazards** in the previous chapter.
- VLSI technology limits the degree up to which pipelining is feasible. (↗ H. Kaeslin. Digital Integrated Circuit Design. Cambridge Univ. Press.).

Hardware Parallelism

Chip area can as well be used for **other types of parallelism**:

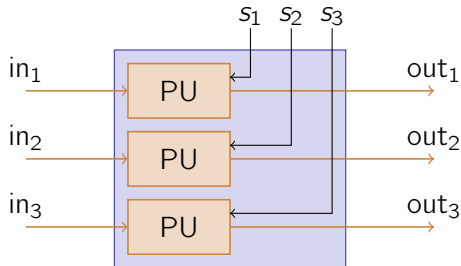


Computer systems typically use identical hardware circuits, but their function may be controlled by different **instruction streams** s_i :



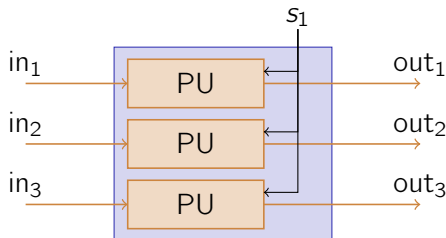
Special Instances (MIMD)

 **Do you know an example of this architecture?**



Special Instances (SIMD)

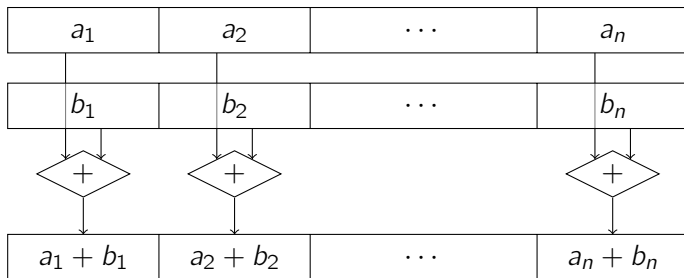
Most modern processors also include a **SIMD** unit:



- Execute same assembly instruction on a set of values.
- Also called **vector unit**; **vector processors** are entire systems built on that idea.

SIMD Programming Model

The processing model is typically based on **SIMD registers** or **vectors**:



Typical values (e.g., x86-64):

- 128 bit-wide registers (`xmm0` through `xmm15`).
- Usable as 16×8 bit, 8×16 bit, 4×32 bit, or 2×64 bit.

- Much of a processor's **control logic** depends on the number of in-flight instructions and/or the number of registers, but **not** on the size of registers.
 - scheduling, register renaming, dependency tracking, ...
- SIMD instructions make **independence** explicit.
 - No data hazards within a vector instruction.
 - Check for data hazards only between vectors.
 - **data parallelism**
- Parallel execution promises n -fold performance advantage.
 - (Not quite achievable in practice, however.)

How can I make use of SIMD instructions as a programmer?

1 Auto-Vectorization

- Some compiler automatically detect opportunities to use SIMD.
- Approach rather limited; don't rely on it.
- Advantage: platform independent

2 Compiler Attributes

- Use `__attribute__((vector_size (...)))` annotations to state your intentions.
- Advantage: platform independent
(Compiler will generate non-SIMD code if the platform does not support it.)

```
/*  
 * Auto vectorization example (tried with gcc 4.3.4)  
 */  
#include <stdlib.h>  
#include <stdio.h>  
  
int  
main (int argc, char **argv)  
{  
    int a[256], b[256], c[256];  
  
    for (unsigned int i = 0; i < 256; i++)  
    {  
        a[i] = i + 1;  
        b[i] = 100 * (i + 1);  
    }  
  
    for (unsigned int i = 0; i < 256; i++)  
        c[i] = a[i] + b[i];  
  
    printf ("c = [ %i, %i, %i, %i ]\n",  
           c[0], c[1], c[2], c[3]);  
  
    return EXIT_SUCCESS;  
}
```

Resulting assembly code (gcc 4.3.4, x86-64):

```
loop:
    movdqu    (%r8,%rcx), %xmm0    ; load a and b
    addl      $1, %esi
    movdqu    (%r9,%rcx), %xmm1    ; into SIMD registers
    paddb     %xmm1, %xmm0         ; parallel add
    movdqa    %xmm0, (%rax,%rcx)    ; write result to memory
    addq      $16, %rcx             ; loop (increment by
    cmpl      %r11d, %esi           ; SIMD length of 16 bytes)
    jnb       loop
```

```

/* Use attributes to trigger vectorization */
#include <stdlib.h>
#include <stdio.h>

typedef int v4si __attribute__((vector_size (16)));

union int_vec {
    int    val[4];
    v4si   vec;
};

typedef union int_vec int_vec;

int
main (int argc, char **argv)
{
    int_vec a, b, c;

    a.val[0] = 1;   a.val[1] = 2;   a.val[2] = 3;   a.val[3] = 4;
    b.val[0] = 100; b.val[1] = 200; b.val[2] = 300; b.val[3] = 400;

    c.vec = a.vec + b.vec;

    printf ("c = [ %i, %i, %i, %i ]\n",
           c.val[0], c.val[1], c.val[2], c.val[3]);

    return EXIT_SUCCESS;
}

```

Resulting assembly code (gcc, x86-64):

```
movl    $1, -16(%rbp)    ; assign constants
movl    $2, -12(%rbp)    ; and write them
movl    $3, -8(%rbp)     ; to memory
movl    $4, -4(%rbp)
movl    $100, -32(%rbp)
movl    $200, -28(%rbp)
movl    $300, -24(%rbp)
movl    $400, -20(%rbp)

movdqa   -32(%rbp), %xmm0 ; load b into SIMD register xmm0
padd     -16(%rbp), %xmm0 ; SIMD xmm0 = xmm0 + a
movdqa   %xmm0, -48(%rbp) ; write SIMD xmm0 back to memory

movl     -40(%rbp), %ecx  ; load c into scalar
movl     -44(%rbp), %edx  ; registers (from memory)
movl     -48(%rbp), %esi
movl     -36(%rbp), %r8d
```

- Data transfers scalar ↔ SIMD go **through memory**.

3 Use C Compiler Intrinsics

- Invoke SIMD instructions directly via **compiler macros**.
- Programmer has good control over instructions generated.
- Code no longer portable to different architecture.
- Benefit (over hand-written assembly): compiler manages register allocation.
- Risk: If not done carefully, automatic glue code (casts, etc.) may make code inefficient.

```

/*
 * Invoke SIMD instructions explicitly via intrinsics.
 */
#include <stdlib.h>
#include <stdio.h>

#include <xmmintrin.h>

int
main (int argc, char **argv)
{
    int a[4], b[4], c[4];
    __m128i x, y;

    a[0] = 1;   a[1] = 2;   a[2] = 3;   a[3] = 4;
    b[0] = 100; b[1] = 200; b[2] = 300; b[3] = 400;

    x = _mm_loadu_si128 ((__m128i *) a);
    y = _mm_loadu_si128 ((__m128i *) b);

    x = _mm_add_epi32 (x, y);

    _mm_storeu_si128 ((__m128i *) c, x);

    printf ("c = [ %i, %i, %i, %i ]\n", c[0], c[1], c[2], c[3]);

    return EXIT_SUCCESS;
}

```

Resulting assembly code (gcc, x86-64):

```
movdqu  -16(%rbp), %xmm1    ; _mm_loadu_si128()
movdqu  -32(%rbp), %xmm0    ; _mm_loadu_si128()
padd    %xmm0, %xmm1        ; _mm_add_epi32()
movdqu  %xmm1, -48(%rbp)    ; _mm_storeu_si128()
```


SIMD and Databases: Scan-Based Tasks

SIMD functionality naturally fits a number of **scan-based** database tasks:


■ arithmetics

```
SELECT price + tax AS net_price  
FROM orders
```

This is what the code examples on the previous slides did.

■ aggregation

```
SELECT COUNT(*)  
FROM lineitem  
WHERE price > 42
```

 How can this be done efficiently?


Similar: SUM(.), MAX(.), MIN(.), ...

Selection queries are a slightly more tricky:

- There are **no branching primitives** for SIMD registers.
 - What would their semantics be anyhow?
- **Moving data** between SIMD and scalar registers is quite **expensive**.
 - Either **go through memory**, move one data item at a time, or extract sign mask from SIMD registers.

Thus:

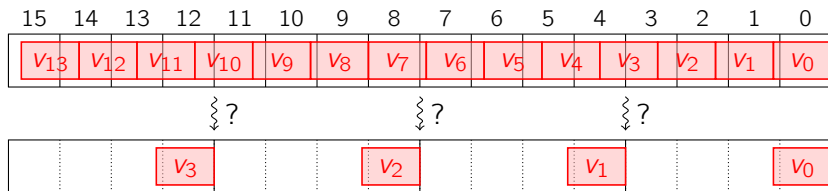
- Use SIMD to generate **bit vector**; interpret it in scalar mode.

 If we can **count** with SIMD, why can't we play the $j += (\dots)$ trick?

Decompression

Column decompression (↗ slides 116ff.) is a good candidate for SIMD optimization.

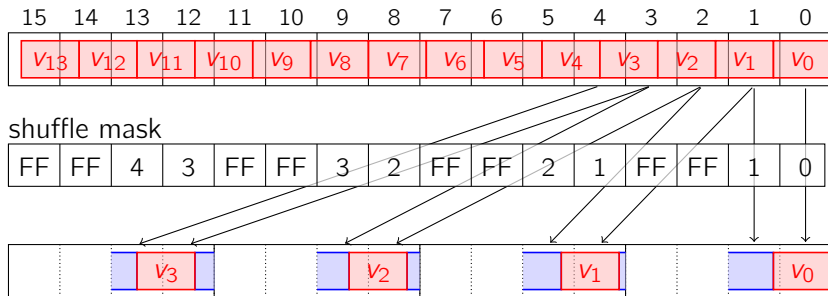
- Use case: n -bit fixed-width **frame of reference** compression; phase 1 (ignore exception values).
 - no branching, no data dependence
- With 128-bit SIMD registers (9-bit compression):



↗ Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *VLDB 2009*.

Decompression—Step 1: Copy Values

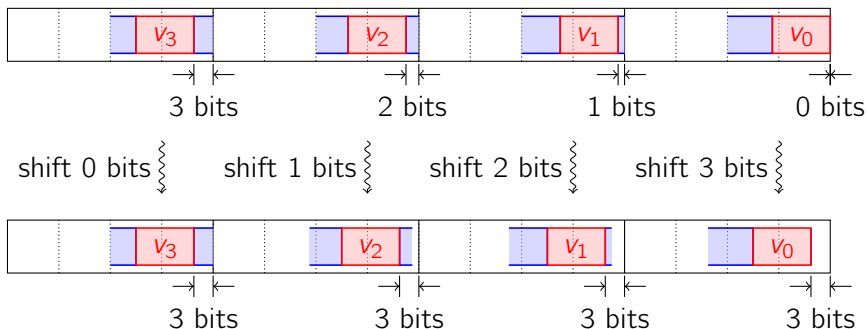
Step 1: Bring data into proper 32-bit words:



- Use **shuffle instructions** to move **bytes** within SIMD registers.
- `__m128i out = _mm_shuffle_epi8(in, shufmask);`

Decompression—Step 2: Establish Same Bit Alignment

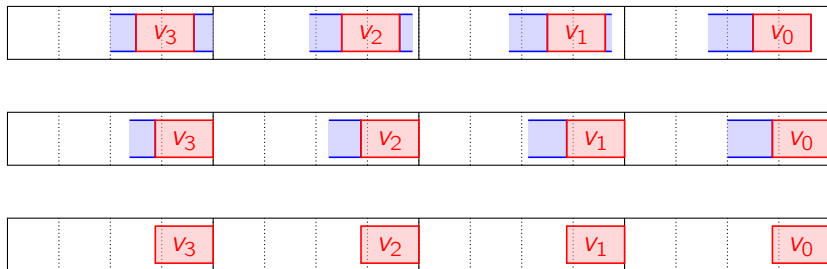
Step 2: Make all four words identically bit-aligned:



SIMD shift instructions do not support variable shift amounts!

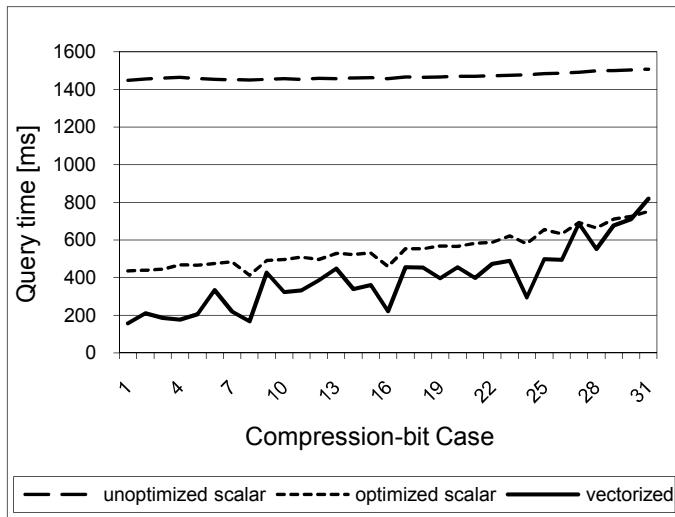
Decompression—Step 3: Shift and Mask

Step 3: Word-align data and mask out invalid bits:



- `__m128i shifted = _mm_srli_epi32(in, 3);`
- `__m128i result = _mm_and_si128(shifted, maskval);`

Decompression Performance



- Time to decompress 1 billion integers (Xeon X5560, 2.8 GHz).

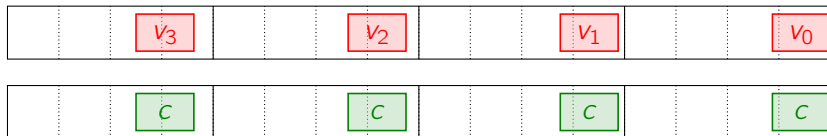
Source: Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009.

- Some SIMD instructions require hard-coded parameters.
Thus: **Expand** code explicitly for all possible values of n .
 - There are at most 32 of them.
 - Fits with operator specialization in column-oriented DBMSs
↗ slide 54
- Loading **constants** into SIMD registers can be relatively expensive (and the number of registers limited).
 - One register for shuffle mask and one register to shift data (step 2) is enough.
- For larger n , a compressed word may span **more than 4 bytes**.
 - Additional tricks needed (shift and blend).

Vectorized Predicate Handling

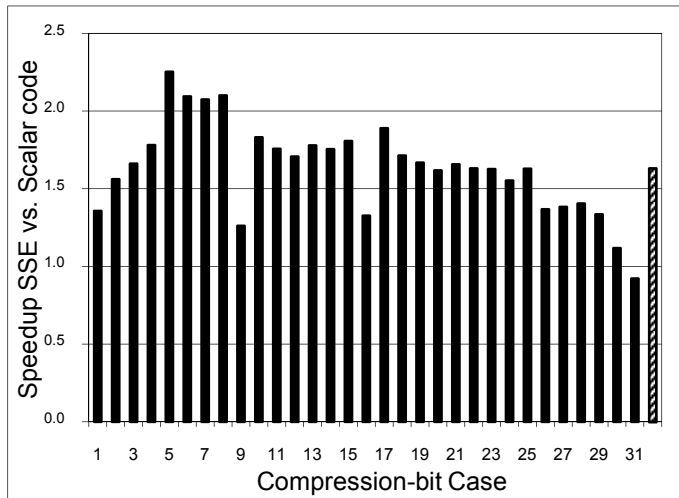
Sometimes it may be sufficient to decompress only partially.

E.g., search queries $v_i < c$:



- Only shuffle and mask (but don't shift).

Vectorized Predicate Handling: Performance



- Speedup versus optimized scalar implementation.

Source: Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB 2009.

Use Case: Tree Search

Another SIMD application: in-memory **tree lookups**.

Base case: **binary tree**, scalar implementation:

```
for (unsigned int i=0; i<n_items; i++) {  
    k=1; /* tree[1] is root node */  
    for (unsigned int lvl=0; lvl<height; lvl++)  
        k=2*k+(item[i]<=tree[k]);  
    result[i]=data[k];  
}
```

- Represent binary tree as array `tree[.]` such that children of n are at positions $2n$ and $2n + 1$.

Can we vectorize the outer loop? **(i.e., find matches for four input items in parallel)**

- Iterations of the outer loop are independent.
- There is no branch in the loop body.

 Current SIMD implementations do not support scatter/gather!



Can we vectorize the inner loop?

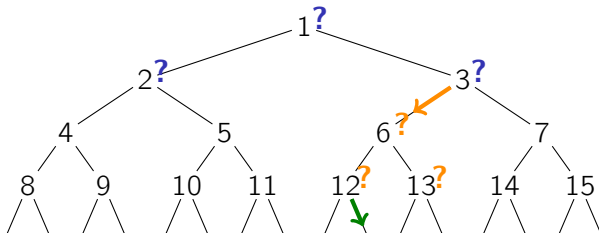
- **Data dependency** between loop iterations (variable k).
- Intuitively: Cannot navigate multiple steps at a time, since first navigation steps are not (yet) known.

But:

- Could **speculatively** navigate levels ahead.

“Speculative” Tree Navigation

Idea: Do comparisons for two levels in parallel.

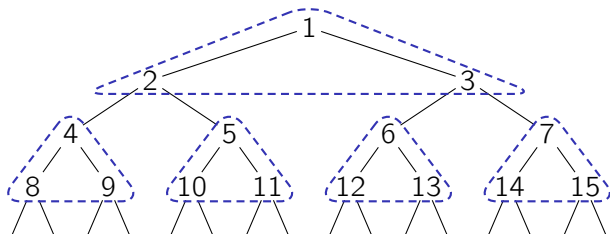


E.g.,

- 1 Compare with nodes 1, 2, and 3 in parallel.
- 2 Follow link to node 6 and compare with nodes 6, 12, and 13.
- 3

SIMD Blocking

Pack tree sub-regions into SIMD registers.

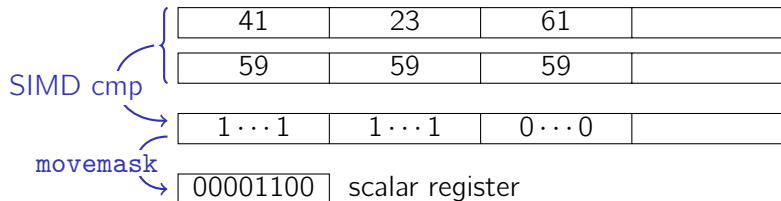
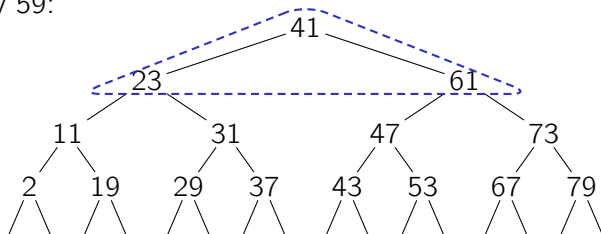


↪ Re-arrange data in memory for this.

↗ Kim *et al.* FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. *SIGMOD 2010*.

SIMD and Scalar Registers

E.g., search key 59:



→ SIMD to compare, scalar to navigate, **movemask** in-between.

Tree Navigation

Use scalar `movemask` result as **index** in **lookup table**:

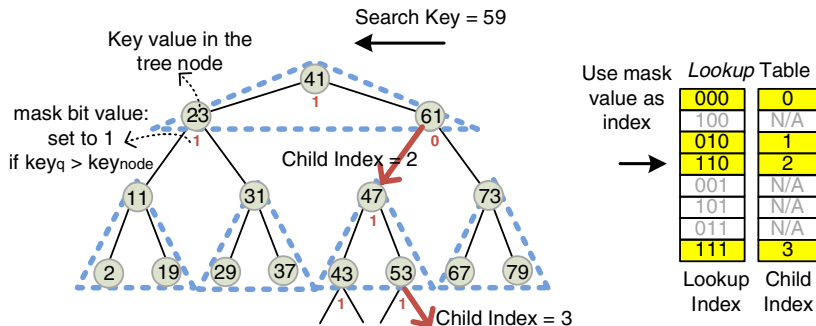


Image source: Kim *et al.* FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. *SIGMOD 2010*.

Hierarchical Blocking

Blocking is a good idea also beyond SIMD.

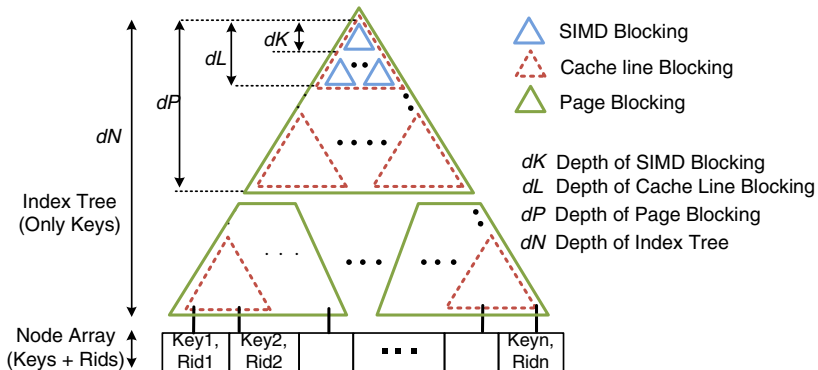
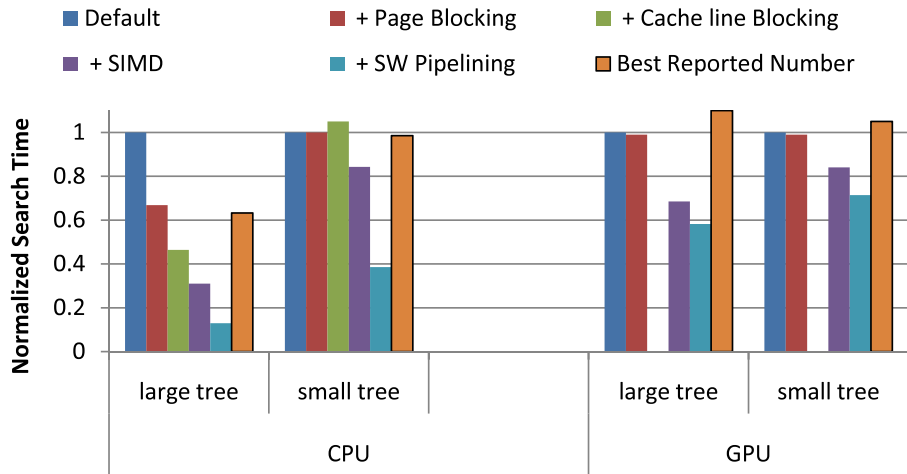


Image source: Kim *et al.* FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. *SIGMOD 2010*.

SIMD Tree Search: Performance



Source: Kim *et al.* FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. *SIGMOD 2010*.