

Programmierpraktikum: Datensysteme

03 Background Query Compilation & Parallelization

Prof. Dr. Matthias Boehm

Technische Universität Berlin

Berlin Institute for the Foundations of Learning and Data

Big Data Engineering (DAMS Lab)

■ #1 Hybrid & Video Recording

- Hybrid lectures (in-person, zoom) with optional attendance

<https://tu-berlin.zoom.us/j/9529634787?pwd=R1ZsN1M3SC9BOU1OcFdmem9zT202UT09>

- Zoom **video recordings**, links from website

https://mboehm7.github.io/teaching/ss25_ppds/index.htm



■ #2 Team Assignment

- 2 teams with 4 and 5 persons, respectively

■ #3 Next Lecture

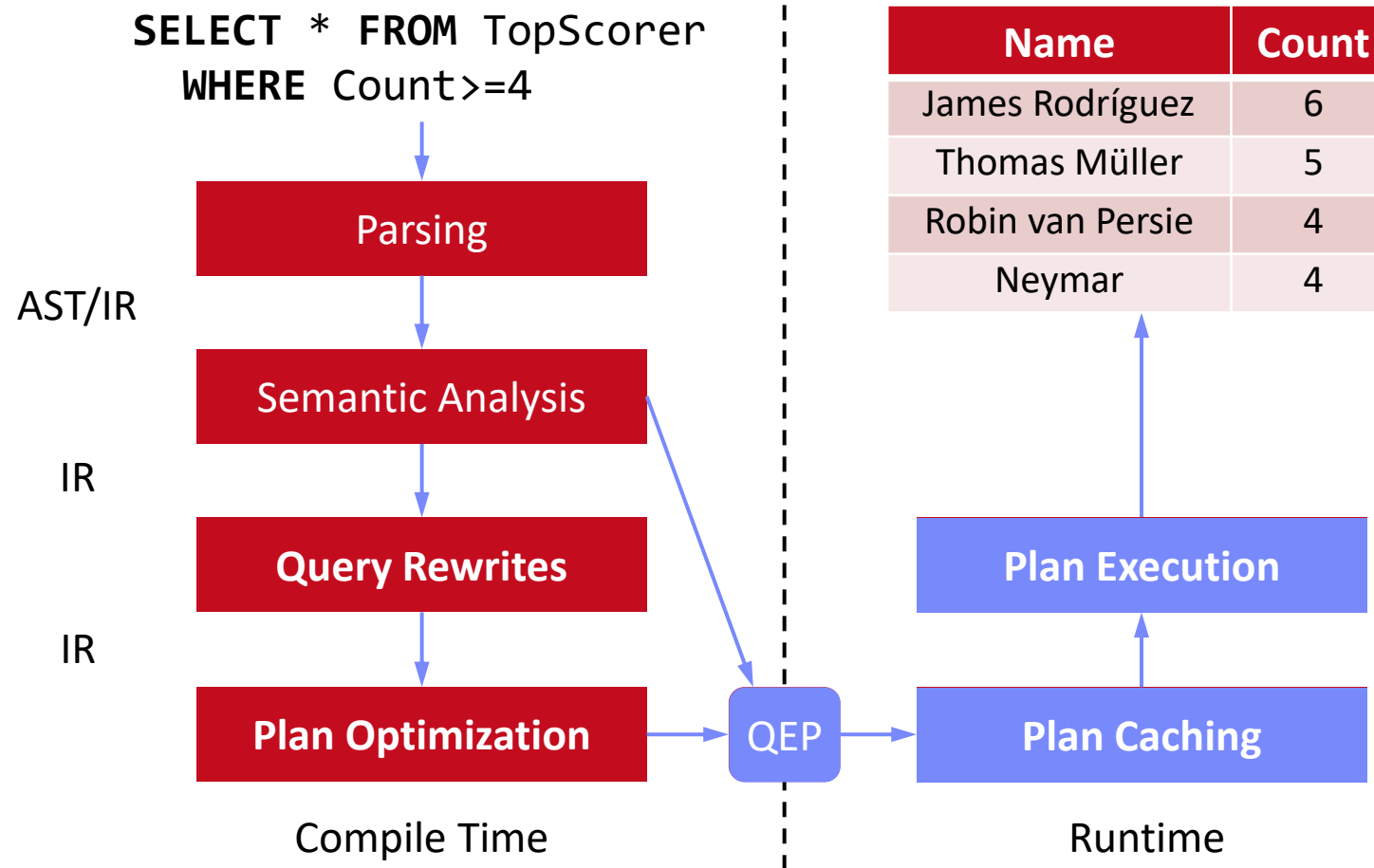
- **May 26, 2025; 4pm** by Christina due to Matthias traveling to Paris

■ #4 Course Achtsamkeits-Meditation

- **8-week course and study, start May 26**
- Mindfulness-Based Stress Reduction, Mental Health, and Productivity
- **Prof. Dr. Maja Adena** (TUB) und Mira Fischer (WZB)
- https://wzb.fra1.qualtrics.com/jfe/form/SV_4IbPvSGunfOVCnk



Recap: Overview Query Processing



Agenda



- **Vectorization and SIMD**
- **Query Compilation**
- **Query Parallelization**

Vectorization and SIMD

SIMD Instruction-level Parallelism (aka Vectorization)
Vectorized Execution Model → Cache-friendly / Auto-SIMD

Terminology Parallelism



■ Flynn's Classification

- SISD, SIMD
- (MISD), MIMD



[Michael J. Flynn, Kevin W. Rudd: Parallel Architectures. ACM Comput. Surv. 28(1) 1996]

■ Example: SIMD Processing

- Streaming SIMD Extensions (SSE)
- Process the same operation on multiple elements at a time (**packed** vs scalar SSE instructions)
- **Data parallelism** (aka: instruction-level parallelism)
- Example: **VFMADD132PD**

	Single Data	Multiple Data
Single Instruction	SISD (uni-core)	SIMD (vector)
Multiple Instruction	MISD (pipelining)	MIMD (multi-core)

2009 Nehalem: **128b** (2xFP64)
2012 Sandy Bridge: **256b** (4xFP64)
2017 Skylake: **512b** (8xFP64)

```
c = _mm512_fmadd_pd(a, b);
```

a								
b								
c								

Background Vector Processors



■ GRAY-1

- 8 x (64 elements x 8B) vector registers
- INT and FP arithmetic @ 80 MHz
- Vector and scalar instructions

[<https://www.computerhistory.org/t dih/september/28/>]

[Richard M. Russell: The CRAY-1 Computer System. CACM 21(1) 1978]



■ NEC Vector Engine v2 20A/20B

- 8/10 vector cores w/ scalar/vector processing units
- Vector width: 256 x 8B = 16,384 bit
- 1.6 GHz, 3.07/6.14 TFLOPs, 1.53 TB/s

[<https://en.wikichip.org/wiki/nec/microarchitectures/sx-aurora>]



@ZAMG



[https://www.zamg.ac.at/cms/en/images/weather/nec/image_view_fullscreen]

■ Others: CPUs, GPUs, FPGAs, DSPs



■ Overview

- Process multiple elements at once
- Avoid conditional branch instructions
- Assuming **column-wise storage**, and vectors of **fixed-sized values**

■ Example Selection

- 16x32b

```
for i = 1 to N step S {  
  Mask[1..S] = SIMD_condition(x[i..i+S-1]);  
  SIMD_Process(Mask[1..S], y[i..i+S-1]); }  
}
```

x	7	1	2	9	3	8	6	7	3	4	9	2	4	5	6	9
mask = x >= 5	1	0	0	1	0	1	1	1	0	0	1	0	0	1	1	1
V = SIMD_bitmap(mask)	77391 // [0, 2 ^(S-1)]															

All match
extraction from y

```
if (V != 0) {  
  for j = 1 to S {  
    tmp = (V >> (S-j)) & 1; /* jth bit */  
    result[pos] = y[j];  
    pos += tmp; } }  
}
```

SIMD Data Processing, cont.

[Jingren Zhou, Kenneth A. Ross:
Implementing database operations using
SIMD instructions. **SIGMOD 2002**]



■ Example Aggregations

- Convert non-matched elements to zero
- Aggregate into vector register, final agg/extraction

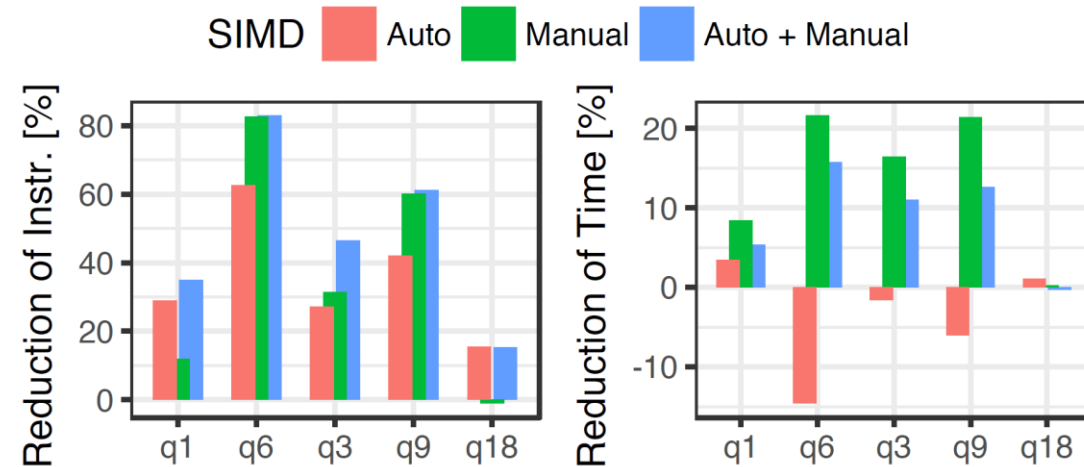
```
SIMD_Process(Mask[1..S], y[1..S]) {  
    temp[1..S] = SIMD_AND( Mask[1..S], y[1..S] );  
    sum[1..S] = SIMD_+( sum[1..S], temp[1..S] );  
}
```

■ Auto Vectorization

- GCC 7.2
- Clang 5.0
- **ICC 18**



[Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, Peter A. Boncz: Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. **PVLDB 11(13) 2018**]



Vectorized Execution (vector-at-a-time)



■ Motivation

- **Iterator Model:** many function calls, no instruction-level parallelism
- **Materialized:** mem-bandwidth-bound

■ Hyper-Pipelining

- Operators work on vectors
- Pipelining of vectors (sub-columns)
- Vector sizes according to cache size
- Pre-compiled function primitives

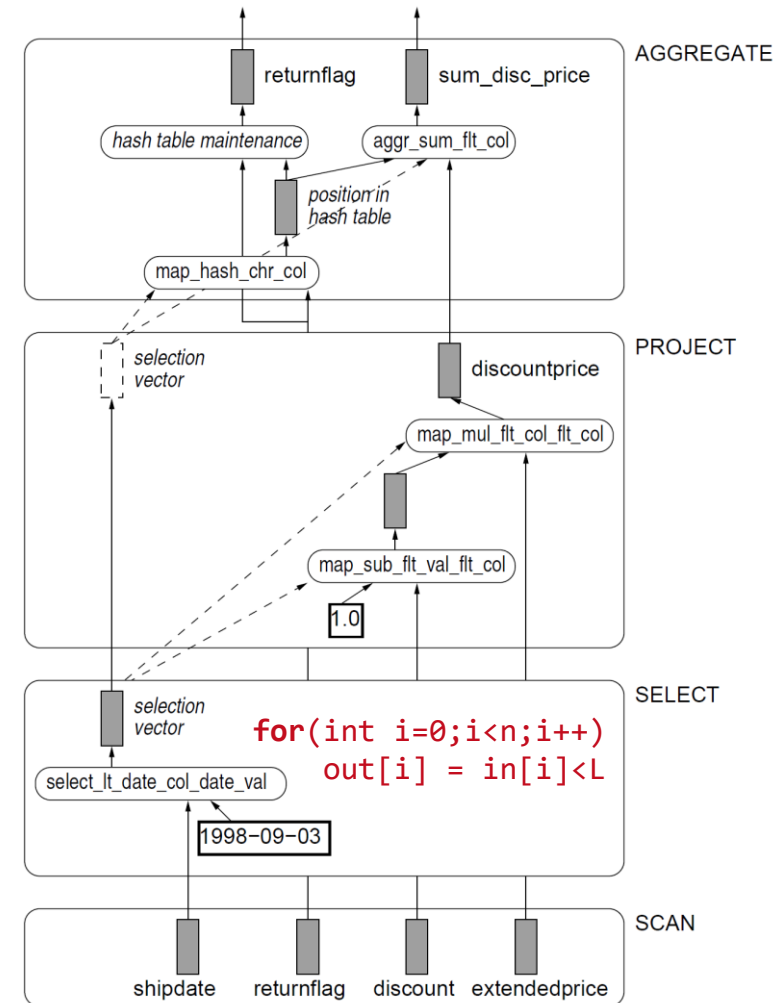
➔ Generalization of execution strategies



[Peter A. Boncz, Marcin Zukowski, Niels Nes:
MonetDB/X100: Hyper-Pipelining Query Execution.
CIDR 2005]



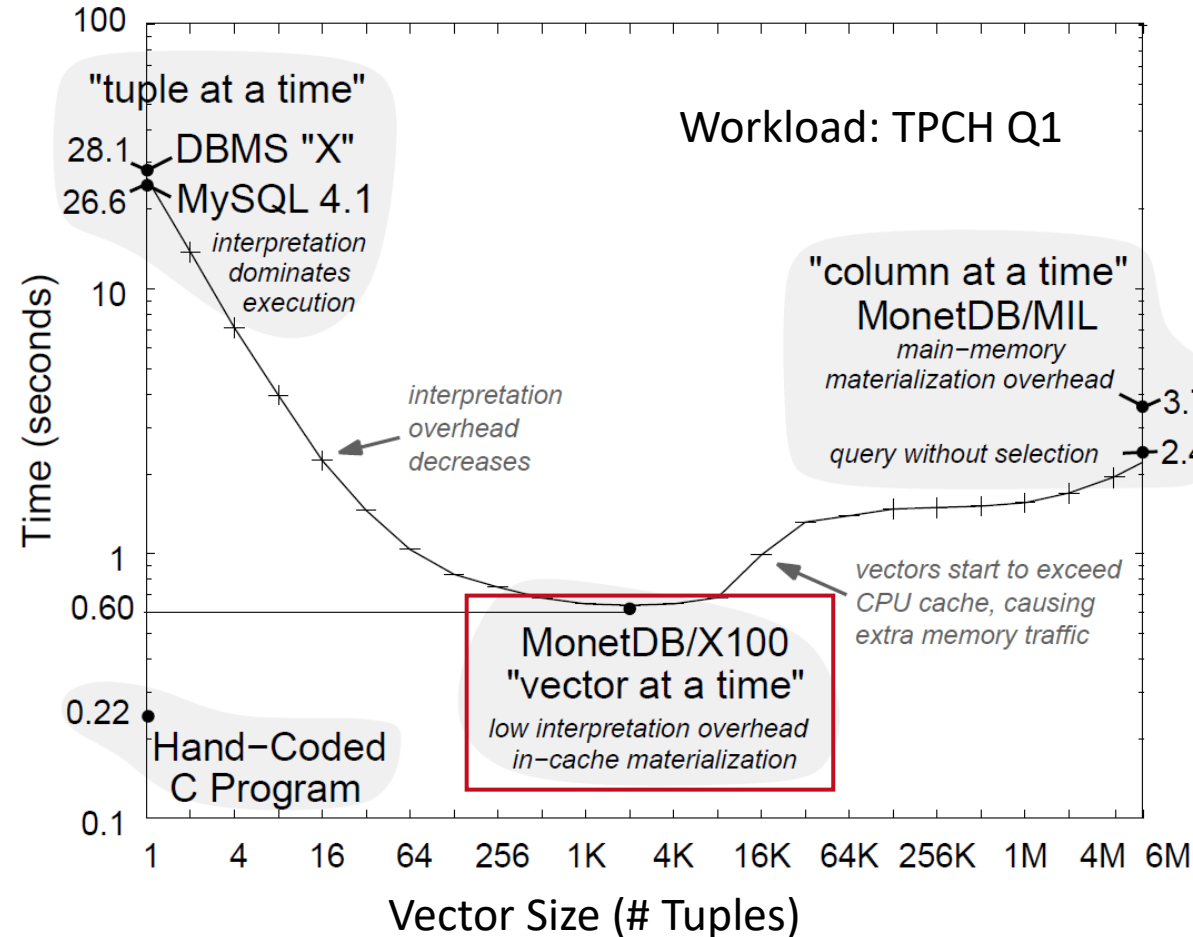
[Marcin Zukowski, Peter A. Boncz, Niels Nes, Sándor Héman:
MonetDB/X100 - A DBMS In The CPU Cache.
IEEE Data Eng. Bull. 28(2), 2005]



Vectorized Execution (vector-at-a-time)



- Idea: Pipelining of vectors (sub columns) s.t. vectors fit in CPU cache hierarchy



Column-oriented storage
Efficient array operations
Memory/cache efficiency
DAG processing
Reuse of intermediates



[Peter A. Boncz, Marcin Zukowski,
Niels Nes: MonetDB/X100: Hyper-
Pipelining Query Execution.
CIDR 2005]

Query Compilation

Holistic Query Evaluation

Data-centric Query Evaluation

Compilation and/or Vectorization

Query Compilation Motivation



■ Background

- Traditional DBMS assume data $\gg$ main memory (I/O dominates)
- Modern in-memory DBMS $\rightarrow$ CPU/memory efficiency crucial

■ Example

```
SELECT sum(price*(1+tax))  
FROM Orders  
WHERE oid >= 100 AND oid <= 200  
GROUPBY category
```



```
for(int i = 0; i < N; i++)  
    if(oid[i] >= 100 && oid[i] <= 200)  
        ret[category] += price[i]*(1+tax[i]);
```

[Juliusz Sompolski, Marcin Zukowski, Peter A. Boncz: Vectorization vs. compilation in query execution. **DaMoN 2011**]



Holistic Query Evaluation

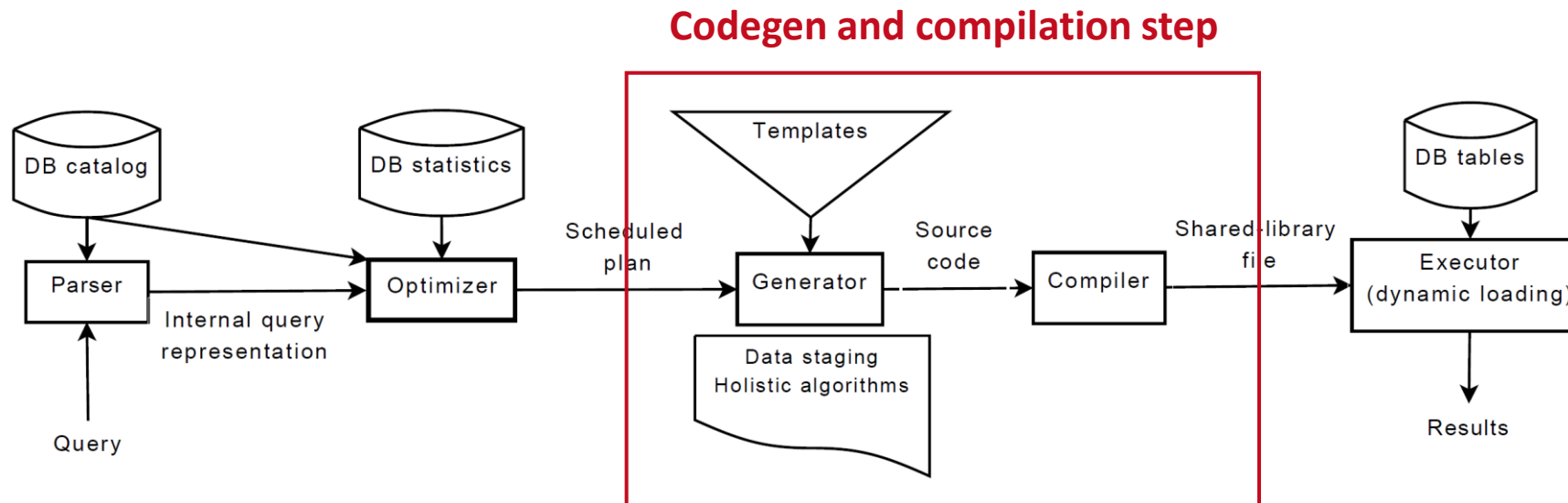
■ Query Processing Architecture

- HIQUE: Holistic Integrated Query Engine
- **Holistic**: Query-awareness + HW-awareness
- Codegen as underlying principle of efficient query evaluation

[Konstantinos Krikellas, Stratis Viglas, Marcelo Cintra: Generating code for holistic query evaluation. **ICDE 2010**]

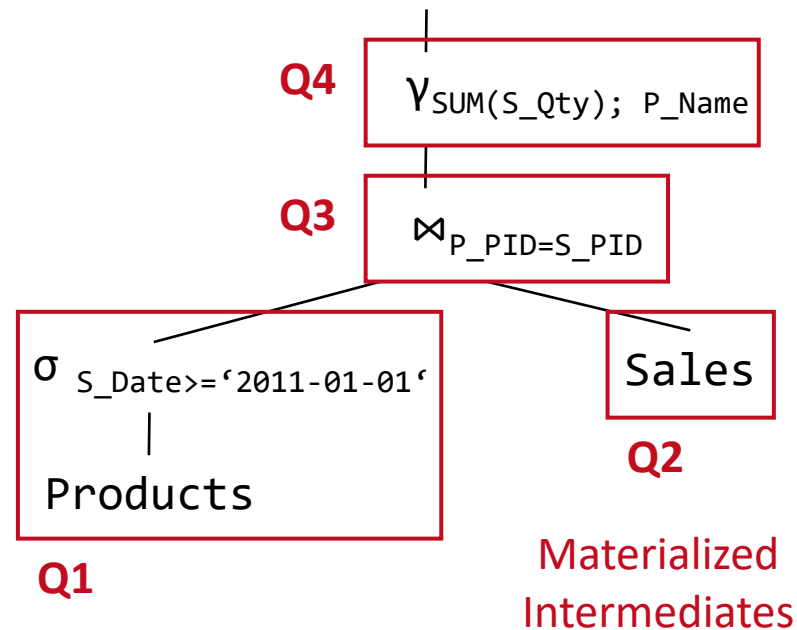


[Konstantinos Krikellas: The case for holistic query evaluation, **PhD Thesis**, University of Edinburgh, **2010**]



■ Code Generation Approach

- #1 **Data Staging**: input tables, selection, projection, pre-processing
- #2 **Holistic Query Instantiation**: join, group-by, order-by



// function prototypes ...

// function implementations

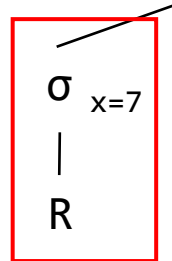
```
Result executeQuery() {
    tmp1 = executeQ1(...);
    tmp2 = executeQ2(...);
    tmp3 = executeQ3(tmp1,tmp2,...);
    return executeQ4(tmp3,...);
}
```

```
... execute01(...) {...}
... execute02(...) {...}
... execute03(...) {...}
... execute04(...) {...}
```

■ Code Generation Approach, cont.

- **Types:** known attribute types → no separate function calls (access, eval)
- **Size:** fixed-length tuples → direct access, cache-conscious blocking
- **Operations:** interleaved operations on cached data

■ #1 Data Staging



Listing 3.2: Type-specific table scan-select

```
1 // loop over pages
2 for (int p = start_page; p <= end_page; p++) {
3     page_struct *page = read_page(p, table);
4     // loop over tuples
5     for (int t = 0; t < page->num_tuples; t++) {
6         void *tuple = page->data + t * tuple_size;
7         int *value = tuple + predicate_offset;
8         if (*value != predicate_value) continue;
9         memcpy(...);
10    }
```

■ #2 Holistic Query Instantiation

- Join Teams
 - Join operators with predicate on **same attribute**
 - Single generated function
- Alternatives
 - Holistic nested loop join
 - Holistic merge join (cooperative staging)
 - Holistic partitioned join
 - Holistic hybrid hash-sort-merge join

Listing 3.6: Generic holistic template for join teams

```
1  /* Code to hash-partition or sort inputs */
2  hash: // examine corresponding partitions together
3  for (k = 0; k < M; k++) {
4  /* update page bounds for all tables, for their k-th partition values */
5  /* sort partitions - only in hybrid hash-sort-merge join */
6
7  for (p_1 = start_page_1; p_1 <= end_page_1; p_1++)
8  page_struct *page_1 = read_page(p_1, partition_1[k]);
9  for (p_2 = start_page_2; p_2 <= end_page_2; p_2++)
10 page_struct *page_2 = read_page(p_2, partition_2[k]);
11 ...
12 for (p_m = start_page_m; p_m <= end_page_m; p_m++) {
13 page_struct *page_m = read_page(p_m, partition_m[k]);
14
15 for (t_1 = 0; t_1 < page_1->num_tuples; t_1++) {
16 void *tuple_1 = page_1->data + t_1 * tuple_size_1;
17 for (t_2 = 0; t_2 < page_2->num_tuples; t_2++) {
18 void *tuple_2 = page_2->data + t_2 * tuple_size_2;
19 int *t1 = tuple_1 + offset_1;
20 int *t2 = tuple_2 + offset_2;
21 if (*t1 != *t2) {
22 merge: // update bounds for all loops
23 continue;
24 }
25 ...
26 for (t_m = 0; t_m < page_m->num_tuples; t_m++) {
27 void *tuple_m = page_m->data + t_m * tuple_size_m;
28 t1 = tuple_1 + offset_1;
29 t2 = tuple_m + offset_m;
30 if (*t1 != *t2) {
31 merge: // update bounds for all loops
32 continue;
33 }
34 add_to_result(tuple_1, ... , tuple_m); /* inlined */
35 } ...}}...}}}
```

M=1 for merge join

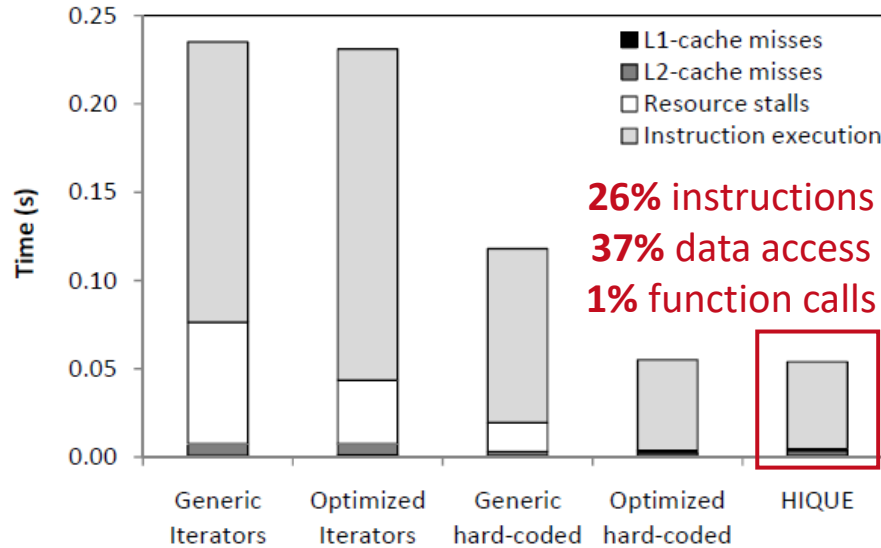
for merge join

for merge join

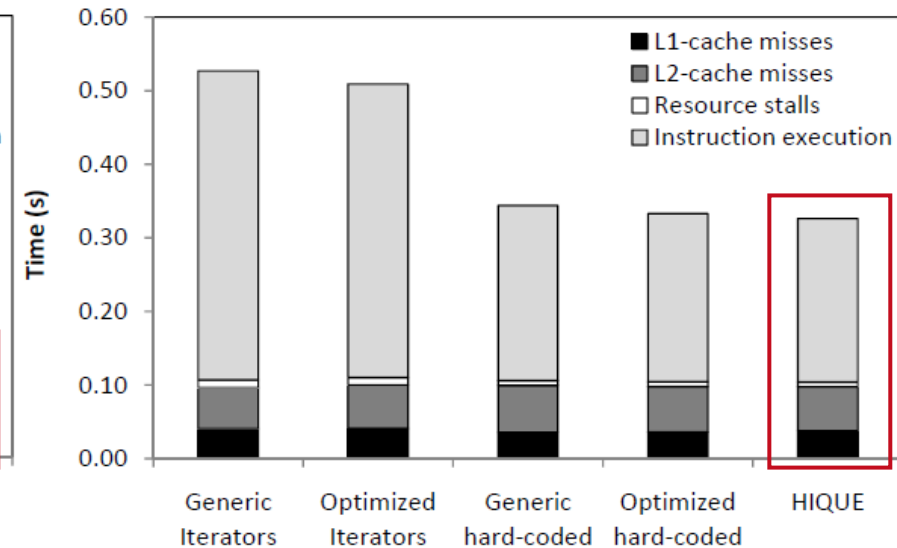
Holistic Query Evaluation, cont.



Runtime Break-Down



(a) Execution time breakdown for Join Query #1



(a) Execution time breakdown for Aggregation Query #1

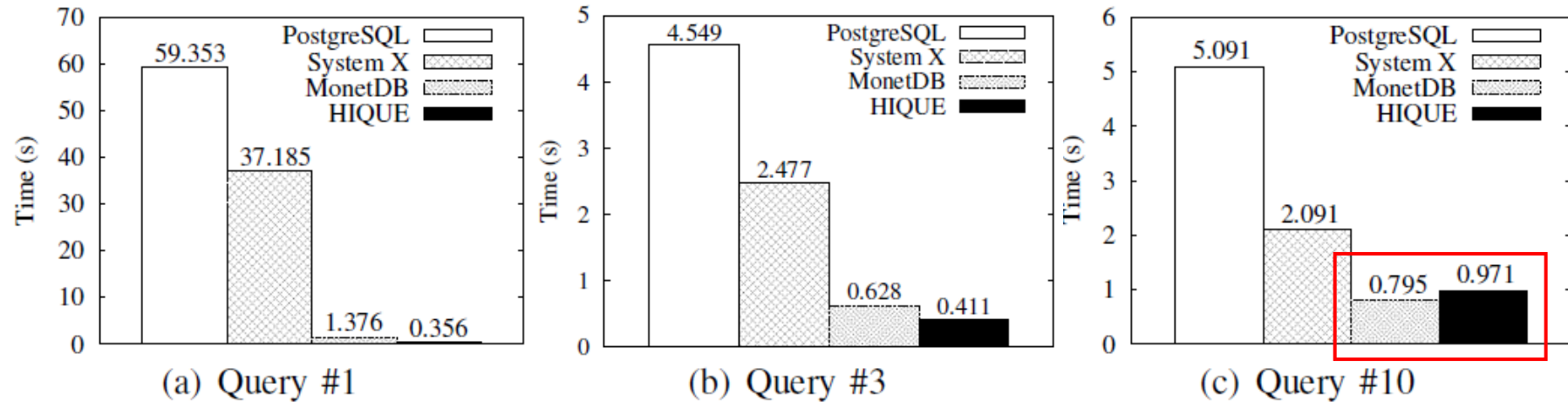
Compiler Optimizations

	Join Query #1		Join Query #2		Aggregation Query #1		Aggregation Query #2	
	-00	-02	-00	-02	-00	-02	-00	-02
Generic iterators	0.802	0.235	1.953	0.995	1.225	0.527	0.136	0.060
Optimized iterators	0.618	0.231	1.850	0.990	1.199	0.509	0.113	0.055
Generic hard-coded	0.430	0.118	1.421	0.688	0.586	0.344	0.095	0.051
Optimized hard-coded	0.267	0.055	1.225	0.622	0.554	0.333	0.080	0.038
HIQUE	0.178	0.054	1.138	0.613	0.543	0.326	0.070	0.033

Holistic Query Evaluation, cont.



Comparison TPC-H Queries



**Compilation time
dominates execution time**

Code Generation Overhead

TPC-H Query	SQL processing (ms)			Compilation (ms)		File sizes (bytes)	
	Parse	Optimize	Generate	with -O0	with -O2	Source	Shared library
#1	21	1	1	121	274	17,733	16,858
#3	11	1	2	160	403	33,795	24,941
#10	15	1	4	213	619	50,718	33,510

Data-centric Query Evaluation

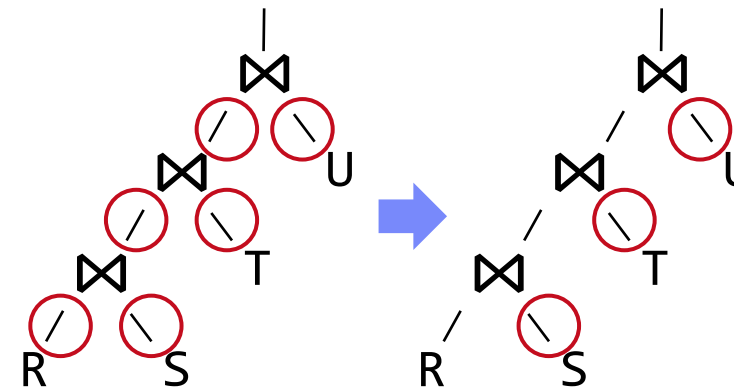
[Thomas Neumann: Efficiently
Compiling Efficient Query Plans for
Modern Hardware. **PVLDB 2011**,
VLDB 2021 Test of Time Award]



■ Motivation

- Algebraic operator model useful for reasoning, but not necessarily a good idea for query processing
- **Code compilation overhead**

Materialized
Intermediates



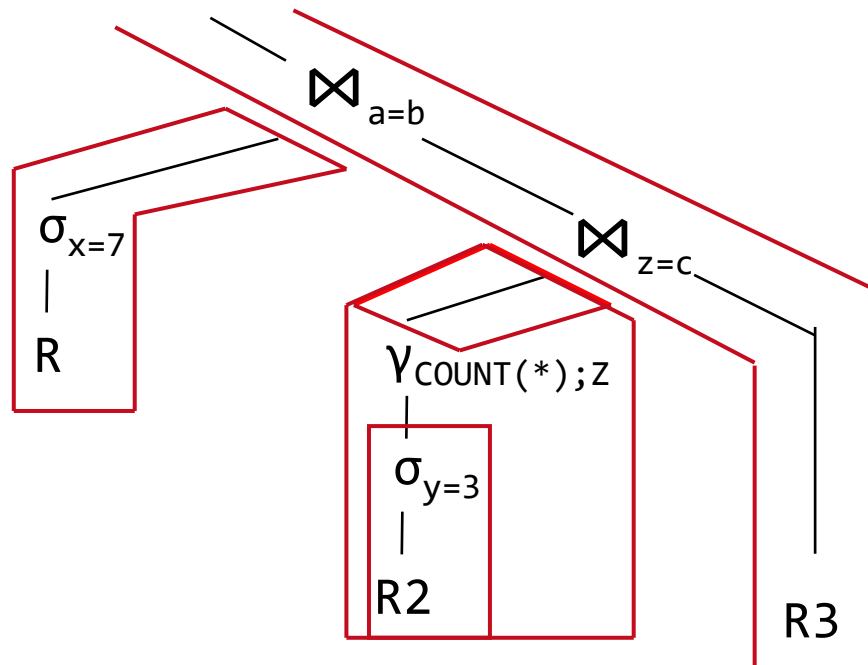
■ Data-centric Processing (not operator-centric)

- Keep **data in CPU registers** as long as possible (no op boundaries)
- Data is pushed towards operators (**code and data locality**)
- Queries are compiled into native machine code using **LLVM**

Data-centric Query Evaluation, cont.

■ Example Plan with Pipeline Boundaries

- Pipeline breaker: op takes a tuple out of register
- Full pipeline breaker: blocking op



```
SELECT * FROM R1,R3,  
  (SELECT R2.z, count(*)  
   FROM R2 WHERE R2.y = 3  
   GROUP BY R2.z) R2  
WHERE R1.x = 7 AND R1.a = R3.b  
AND R2.z = R3.c
```

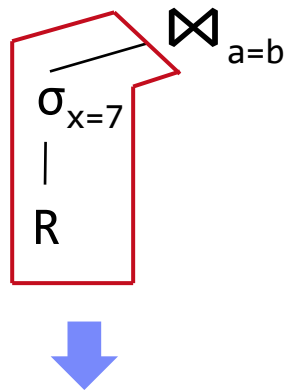
Compiled Query (not LLVM)

```
initialize memory of  $\bowtie_{a=b}$ ,  $\bowtie_{z=c}$ , and  $\Gamma_z$   
for each tuple  $t$  in  $R_1$   
  if  $t.x = 7$   
    materialize  $t$  in hash table of  $\bowtie_{a=b}$   
for each tuple  $t$  in  $R_2$   
  if  $t.y = 3$   
    aggregate  $t$  in hash table of  $\Gamma_z$   
for each tuple  $t$  in  $\Gamma_z$   
  materialize  $t$  in hash table of  $\bowtie_{z=c}$   
for each tuple  $t_3$  in  $R_3$   
  for each match  $t_2$  in  $\bowtie_{z=c}[t_3.c]$   
    for each match  $t_1$  in  $\bowtie_{a=b}[t_3.b]$   
      output  $t_1 \circ t_2 \circ t_3$ 
```

■ Data-Centric Operator Model

- Conceptual data-centric operator model, used during compilation
- #1 **produce()**: produce result tuples (top-down)
- #2 **consume(attributes, source)**: receive input tuples (bottom-up)

■ Example



```
for each tuple  $t$  in  $R_1$ 
  if  $t.x = 7$ 
    materialize  $t$  in hash table of  $\Join_{a=b}$ 
```

$\Join$.produce	$\Join$.left.produce; $\Join$.right.produce;
$\Join$.consume(a,s)	if ($s==\Join$.left) print "materialize tuple in hash table"; else print "for each match in hashtable[" $+a.joinattr+"$]; $\Join$.parent.consume($a+new\ attributes$)
σ .produce	σ .input.produce
σ .consume(a,s)	print "if " + σ .condition; σ .parent.consume(attr, σ)
scan.produce	print "for each tuple in relation" scan.parent.consume(attributes,scan)

■ Example

LLVM Fragment:

$Y_{\text{COUNT}}(*); Z(\sigma_{y=3}(R2))$

```
define internal void @scanConsumer(%8* %executionState, %Fragment_R2* %data) {
body:
    ...
    %columnPtr = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 0
    %column = load i32** %columnPtr, align 8
    %columnPtr2 = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 1
    %column2 = load i32** %columnPtr2, align 8
    ... (loop over tuples, currently at %id, contains label %cont17)
    %yPtr = getelementptr i32* %column, i64 %id
    %y = load i32* %yPtr, align 4
    %cond = icmp eq i32 %y, 3
    br i1 %cond, label %then, label %cont17
then:
    %zPtr = getelementptr i32* %column2, i64 %id
    %z = load i32* %zPtr, align 4
    %hash = urem i32 %z, %hashTableSize
    %hashSlot = getelementptr %"HashGroupify::Entry"* %hashTable, i32 %hash
    %hashIter = load %"HashGroupify::Entry"* %hashSlot, align 8
    %cond2 = icmp eq %"HashGroupify::Entry"* %hashIter, null
    br i1 %cond2, label %loop20, label %else26
    ... (check if the group already exists, starts with label %loop20)
else26:
    %cond3 = icmp le i32 %spaceRemaining, i32 8
    br i1 %cond3, label %then28, label %else47
    ... (create a new group, starts with label %then28)
else47:
    %ptr = call i8* @ZN12HashGroupify15storeInputTupleEmj
        (%"HashGroupify"* %1, i32 hash, i32 8)
    ... (more loop logic)
}
```

1. locate tuples in memory

2. loop over all tuples

3. filter $y = 3$

4. hash z

5. lookup in hash table (C++ data structure)

6. not found, check space

7. full, call C++ to allocate mem or spill

Data-centric Query Evaluation, cont.



■ Experiments

- TPC-CH (extended TPC-C+TPC-H)

	HyPer + C++	HyPer + LLVM
TPC-C [tps]	161,794	169,491
total compile time [s]	16.53	0.81

	Q1	Q2	Q3	Q4	Q5
HyPer + C++ [ms]	142	374	141	203	1416
compile time [ms]	1556	2367	1976	2214	2592
HyPer + LLVM	35	125	80	117	1105
compile time [ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	16410	3830	15212

■ Code Quality

- Instruction cache misses (L1i)
- Data cache misses (L1d, L2d)

	Q1		Q2		Q3		Q4		Q5	
	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
branches	19,765,048	144,557,672	37,409,113	114,584,910	14,362,660	127,944,656	32,243,391	408,891,838	11,427,746	333,536,532
mispredicts	188,260	456,078	6,581,223	3,891,827	696,839	1,884,185	1,182,202	6,577,871	639	6,726,700
I1 misses	2,793	187,471	1,778	146,305	791	386,561	508	290,894	490	2,061,837
D1 misses	1,764,937	7,545,432	10,068,857	6,610,366	2,341,531	7,557,629	3,480,437	20,981,731	776,417	8,573,962
L2d misses	1,689,163	7,341,140	7,539,400	4,012,969	1,420,628	5,947,845	3,424,857	17,072,319	776,229	7,552,794
I refs	132 mil	1,184 mil	313 mil	760 mil	208 mil	944 mil	282 mil	3,140 mil	159 mil	2,089 mil

Other Systems w/ Query Compilation



[<http://sites.computer.org/debull/A14mar/issue1.htm>]

■ IEEE Data Engineering Bulletin

March 2014

Vol. 37 No. 1



IEEE Computer Society

Letters

Letter from the Editor-in-Chief	David Lomet	1
Letter from the Special Issue Editor	S. Sudarshan	2

Special Issue on When Compilers Meet Database Systems

HyPer	Compiling Database Queries into Machine Code	Thomas Neumann and Viktor Leis	3
HIQUE	Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes	Stratis D. Viglas, Gavin Bierman and Fabian Nagel	12
Hekaton	Compilation in the Microsoft SQL Server Hekaton Engine	Craig Freedman, Erik Ismert, and Per-Ake Larson	22
Impala	Runtime Code Generation in Cloudera Impala	Skye Wanderman-Milne and Nong Li	31
	Database Application Developer Tools Using Static Analysis and Dynamic Profiling	Surajit Chaudhuri, Vivek Narasayya and Manoj Syamala	38
	Using Program Analysis to Improve Database Applications	Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden and Andrew C. Myers	48
	Database-Aware Program Optimization via Static Analysis	Karthik Ramachandra and Ravindra Guravannavar	60
LegoBase/ ScaLite	Abstraction Without Regret in Database Systems Building: a Manifesto	Christoph Koch	70

■ Improved Branch Prediction

- No-branch:
pos+=pred(data[i])
- Hash table lookup



[Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. **PVLDB 2011**]

```
Entry* iter=hashTable[hash];  
while (iter) {  
    ... // inspect the entry  
    iter=iter->next;  
}
```



```
existing entry (~ true)  
Entry* iter=hashTable[hash];  
if (iter) do {  
    ... // inspect the entry  
    iter=iter->next;  
} while (iter);  
end of chain (~ false)
```

■ SIMD Loop Tiling and Fission

- Loop tiling (vectorization) for SIMD
- Loop fission into parallel and serial ops



[Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, Stan Zdonik: An Architecture for Compiling UDF-centric Workflows. **PVLDB 8(12) 2015**]

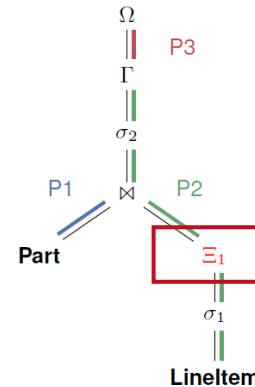
```
data[N]; hash[TILE]; sum[M] = {0};  
for (i = 0; i < N / TILE; i++) {  
    offset = i * TILE;  
    for (j = 0; j < TILE; j++) {  
        key = k(data[offset + j]);  
        hash[j] = h(key);  
    }  
    for (j = 0; j < TILE; j++)  
        sum[hash[j]] += data[offset + j];  
}
```

Compilation w/ SIMD, Prefetch, Decompress



■ Relaxed Operator Fusion (ROF)

- Introduce buffered stage boundary for **vectorized execution**
- SIMD operations after boundary (w/ repacking after SIMD ops)
- Prefetching before boundary



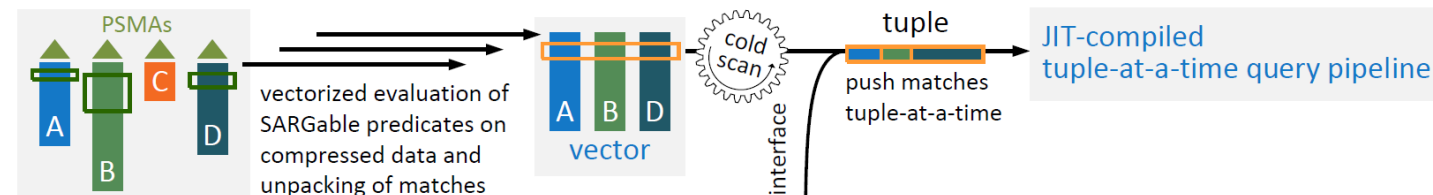
[Prashanth Menon, Andrew Pavlo, Todd C. Mowry: Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. **PVLDB 11(1) 2017**]



■ Data Blocks

- Hot and cold (compressed) blocks
- SIMD predicated evaluation on blocks, output unpacking into **vectors** of 8192 tuples
- Vector tuples fed into JIT-compiled pipelines

[Harald Lang et al: Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. **SIGMOD 2016**]



Compilation vs Vectorized Execution

[Timo Kersten et al.: Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. **PVLDB 11(13) 2018**]

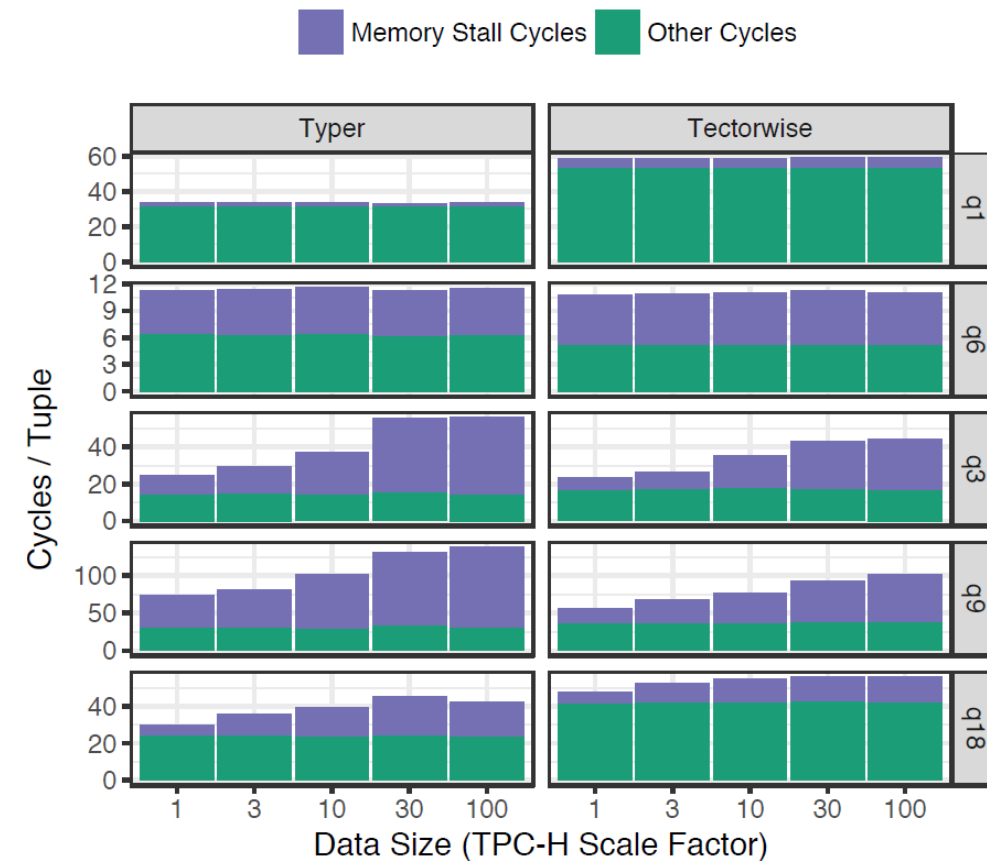
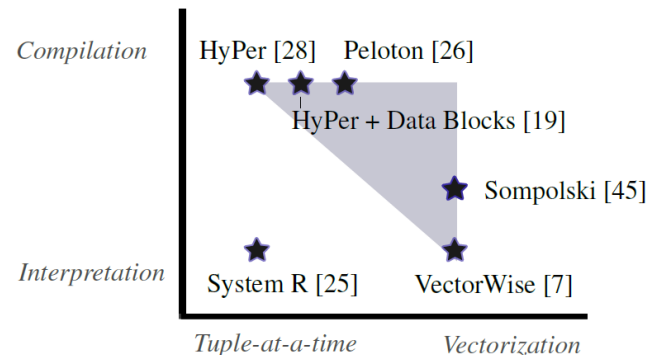


■ Motivation

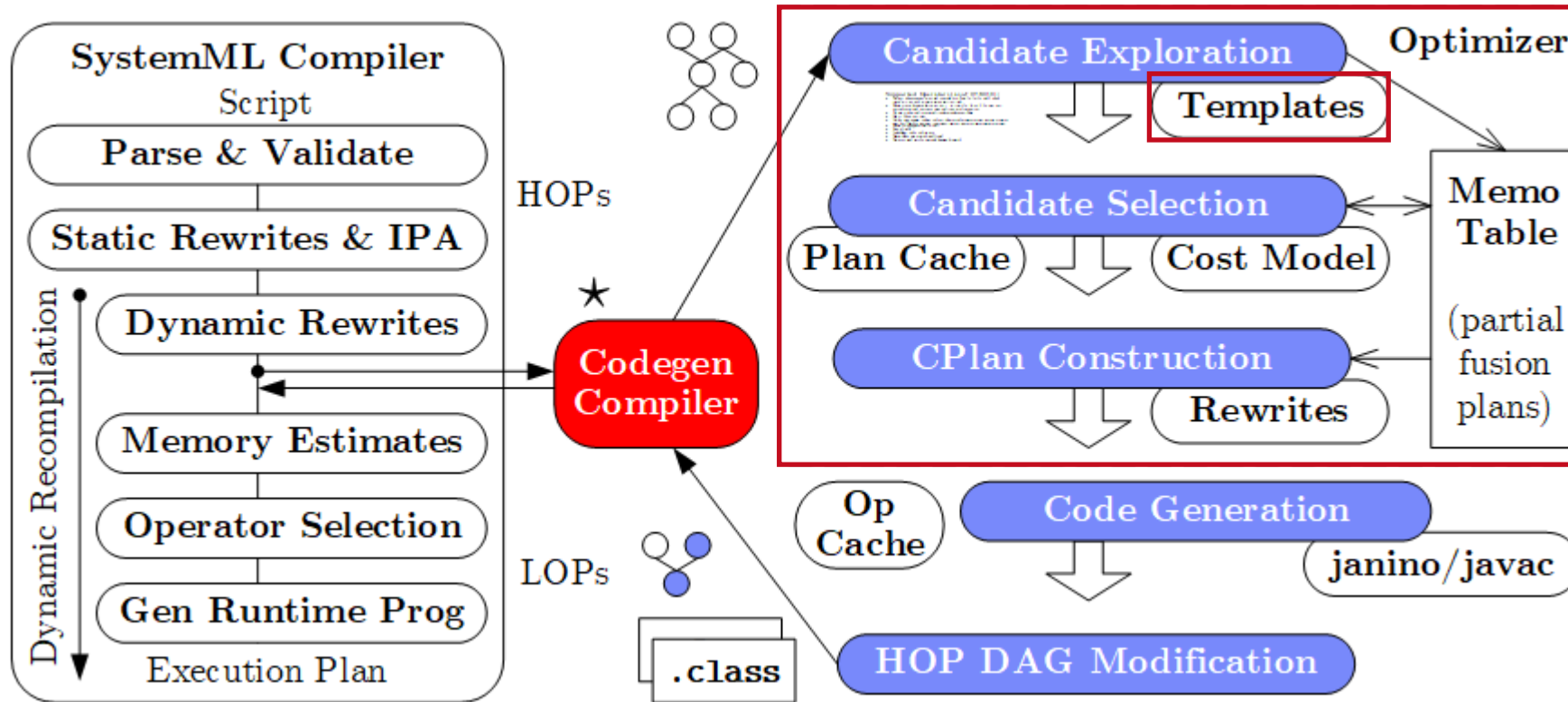
- **SotA**: compilation or vectorization
- **Typer**: Test data-centric query eval (HyPer)
- **Tectorwise**: Test vectorized eval (VectorWise)

■ Selected Results

- Neither system clearly dominated
- Both with large differences to others



Practical, exact, cost-based optimizer



Templates:
Cell, Row, MAgg, Outer
w/ different data bindings



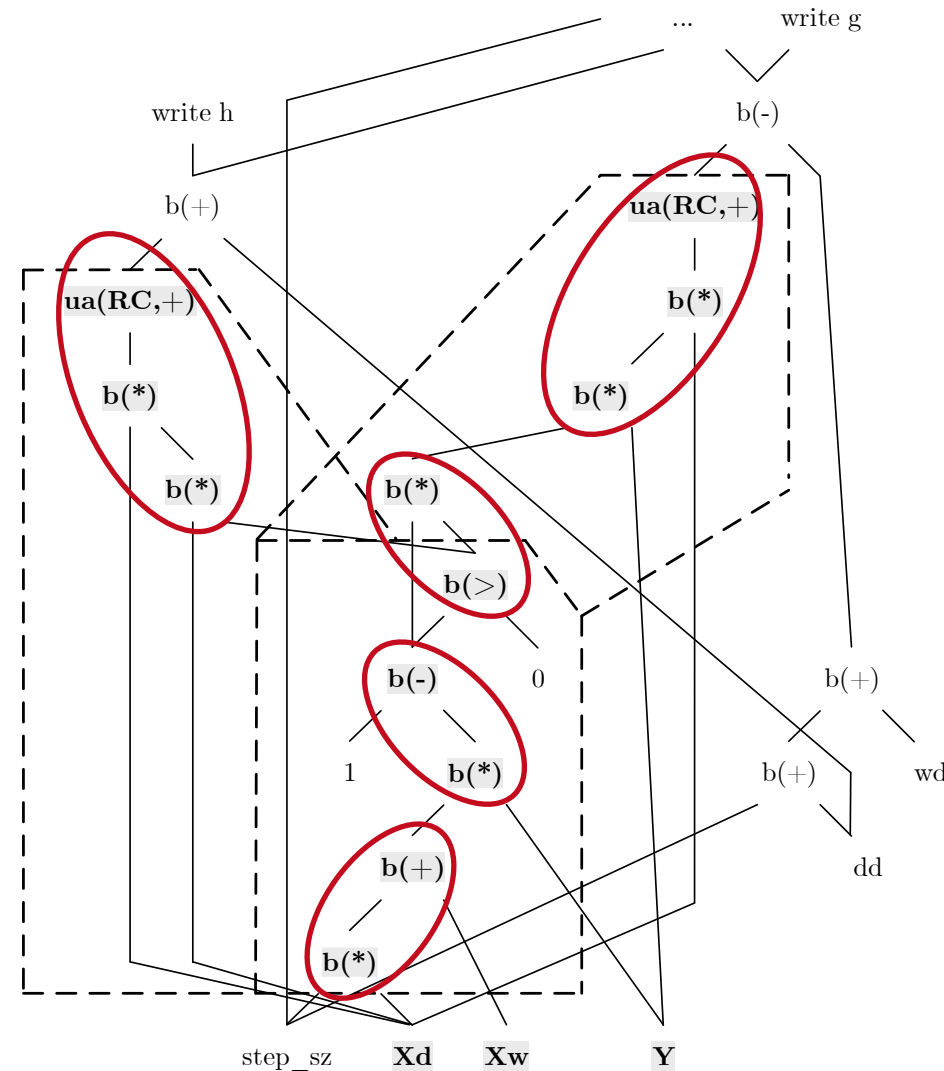
CPlan representation and codegen similar in TF XLA (HLO primitives, pre-clustering of nodes, caching, LLVM codegen)

■ L2SVM Inner Loop

```
1: while(continueOuter & iter < maxi) {  
2   #...  
3:   while(continueInner) {  
4:     out = 1-Y* (Xw+step_sz*Xd);  
5:     sv = (out > 0);  
6:     out = out * sv;  
7:     g = wd + step_sz*dd  
        - sum(out * Y * Xd);  
8:     h = dd + sum(Xd * sv * Xd);  
9:     step_sz = step_sz - g/h;  
10:  }} ...
```

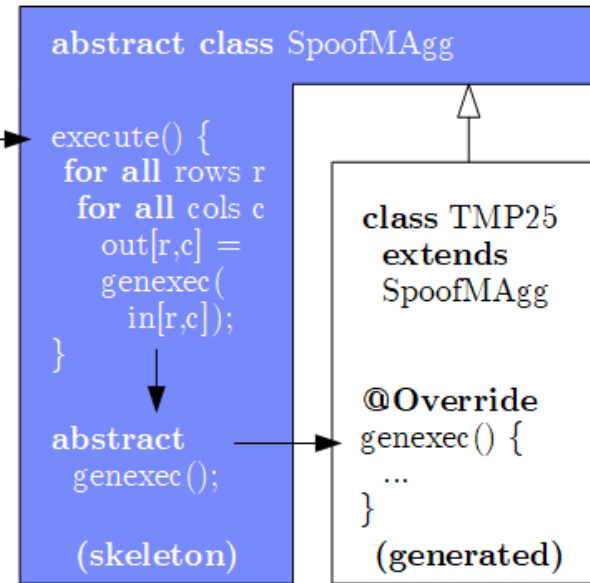
■ # of Vector Intermediates

- Base (w/o fused ops): **10**
- Fused (w/ fused ops): **4**



■ Template Skeleton

- Data access (dense, sparse, compressed)
- Cache blocking
- Multi-threading
- Final aggregation



■ # of Vector Intermediates

- Gen (codegen ops): 0

```
public final class TMP25 extends SpoofMAgg {
    public TMP25() {
        super(false, AggOp.SUM, AggOp.SUM);
    }
    protected void genexec(double a, SideInput[] b,
        double[] scalars, double[] c, ...) {
        double TMP11 = getValue(b[0], rowIndex);
        double TMP12 = getValue(b[1], rowIndex);
        double TMP13 = a * scalars[0];
        double TMP14 = TMP12 + TMP13;
        double TMP15 = TMP11 * TMP14;
        double TMP16 = 1 - TMP15;
        double TMP17 = (TMP16 > 0) ? 1 : 0;
        double TMP18 = a * TMP17;
        double TMP19 = TMP18 * a;
        double TMP20 = TMP16 * TMP17;
        double TMP21 = TMP20 * TMP11;
        double TMP22 = TMP21 * a;
        c[0] += TMP19;
        c[1] += TMP22;
    }
}
```

Excursus: SystemDS Codegen – Example MLogreg (Row)

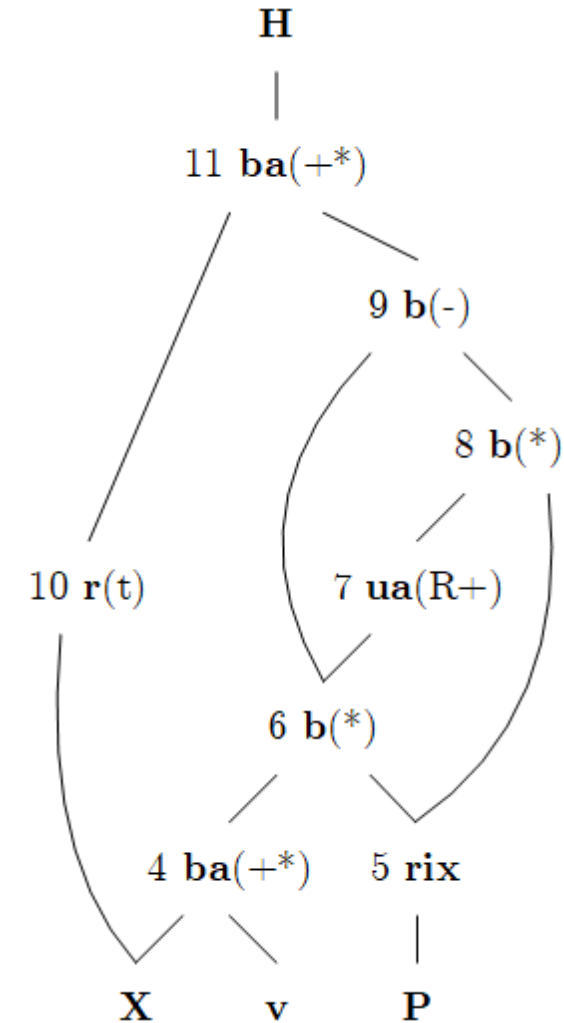


■ MLogreg Inner Loop (main expr on feature matrix X)

```
1: Q = P[, 1:k] * (X %%% v)
2: H = t(X) %%% (Q - P[, 1:k] * rowSums(Q))
```

```
public final class TMP25 extends SpoofRow {
  public TMP25() {
    super(RowType.COL_AGG_B1_T, true, 5);
  }
  protected void genexecDense(double[] a, int ai,
    SideInput[] b, double[] c, ..., int len) {
    double[] TMP11 = getVector(b[1].vals(rix),...);
    double[] TMP12 = vectMatMult(a, b[0].vals(rix),...);
    double[] TMP13 = vectMult(TMP11, TMP12, 0, 0,...);
    double TMP14 = vectSum(TMP13, 0, TMP13.length);
    double[] TMP15 = vectMult(TMP11, TMP14, 0,...);
    double[] TMP16 = vectMinus(TMP13, TMP15, 0, 0,...);
    vectOuterMultAdd(a, TMP16, c, ai, 0, 0,...);
  }
  protected void genexecSparse(double[] avals, int[] aix,
    int ai, SideInput[] b, ..., int len) {...}
}
```

“vectorized
row ops”



Query Parallelization

Intra- and Inter-Operator Parallelism

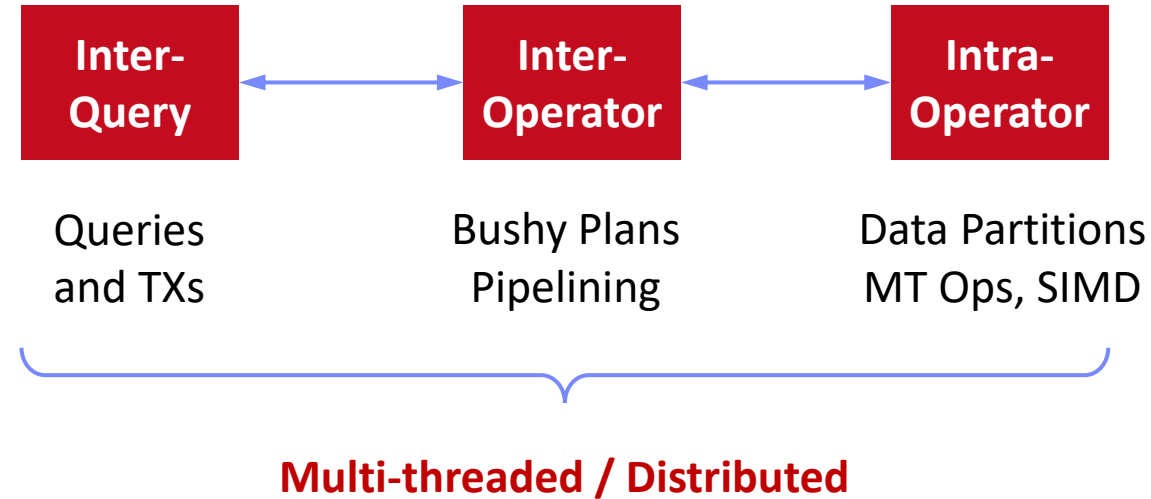
Fine-grained Pipeline Parallelism

Workload Management / Inter-Query Parallelism

Overview Query Parallelism



■ Types of Parallelism



■ Beware: Danger of Interference

- #1 Locks and latches on hot data items → increasing TX aborts
 - #2 Temporary memory/IO requirements (see [buffer pool management](#))
 - #3 CPU and cache interference (e.g., context switches)
 - #4 Throughput vs latency vs freshness vs fairness vs priorities
- ➔ Dedicated [DB workload management](#) & [DB schedulers](#)

Recap: Iterator Model Overview

[Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. IEEE Trans. Knowl. Data Eng. 1994]



Volcano Iterator Model

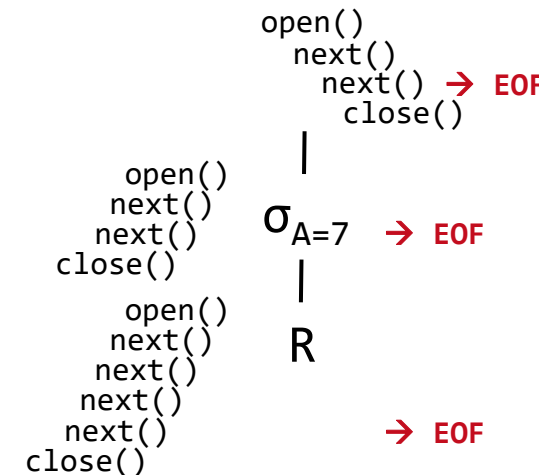
- Open-Next-Close (ONC) interface
- Query execution from root node (pull-based) → Pipelined

Scalable (small memory)
High CPI measures

Example

$\sigma_{A=7}(R)$

```
void open() { R.open(); }  
void close() { R.close(); }  
Record next() {  
    while( (r = R.next()) != EOF )  
        if( p(r) ) //A==7  
            return r;  
    return EOF;  
}
```



Blocking Operators

- Sorting, grouping/aggregation, build-phase of (simple) hash joins

PostgreSQL: `Init()`,
`GetNext()`, `ReScan()`, `MarkPos()`,
`RestorePos()`, `End()`

Intra- and Inter-Operator Parallelism

[Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. **SIGMOD 1990**]

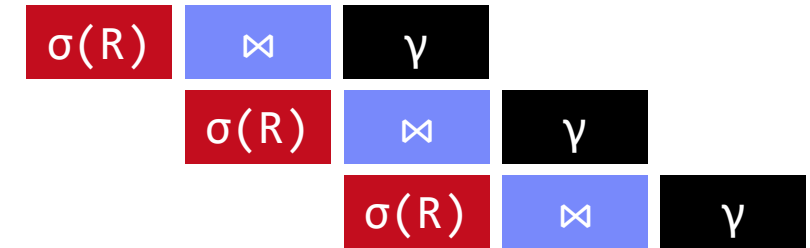
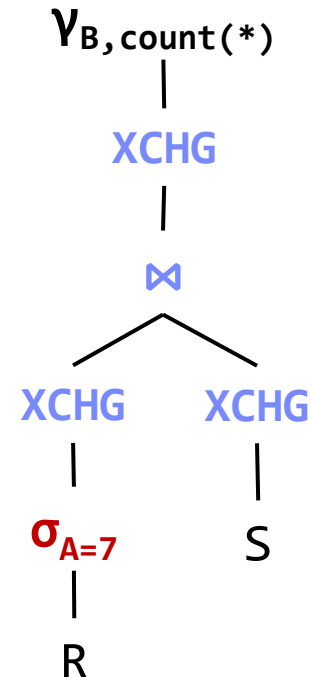


■ Overview

- Seamless parallelization in iterator model via dedicated **exchange** operator
- Avoid unnecessary overhead for local subplans

■ Inter-Operator Parallelism

- **Vertical parallelism** in terms of pipelining
- **Open:** create new process
- **Next:** transfer packets of records (1 .. 32,000)
- **Close:** shutdown child processes



Intra- and Inter-Operator Parallelism, cont.

[Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. **SIGMOD 1990**]

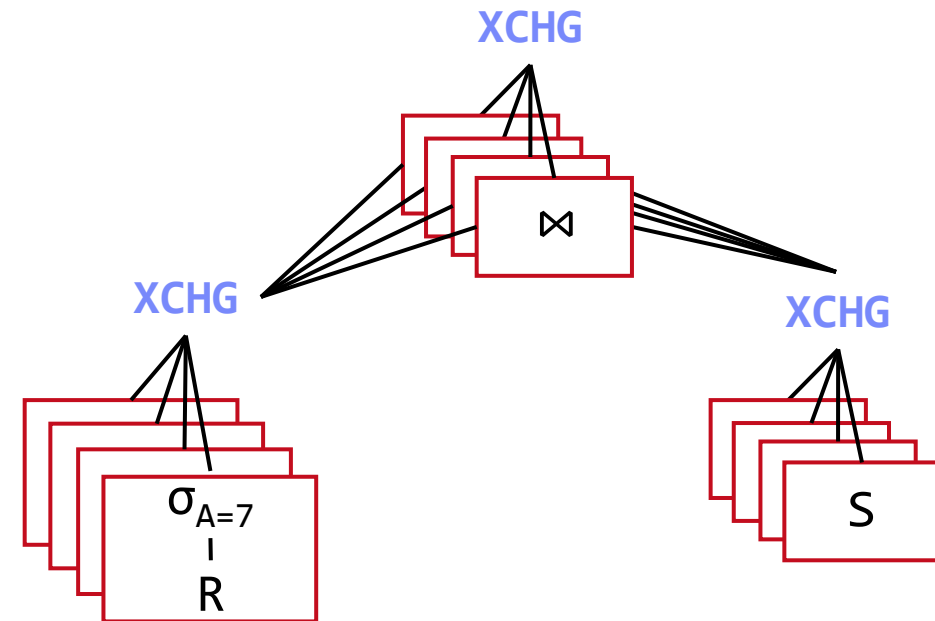


■ Intra-Operator Parallelism

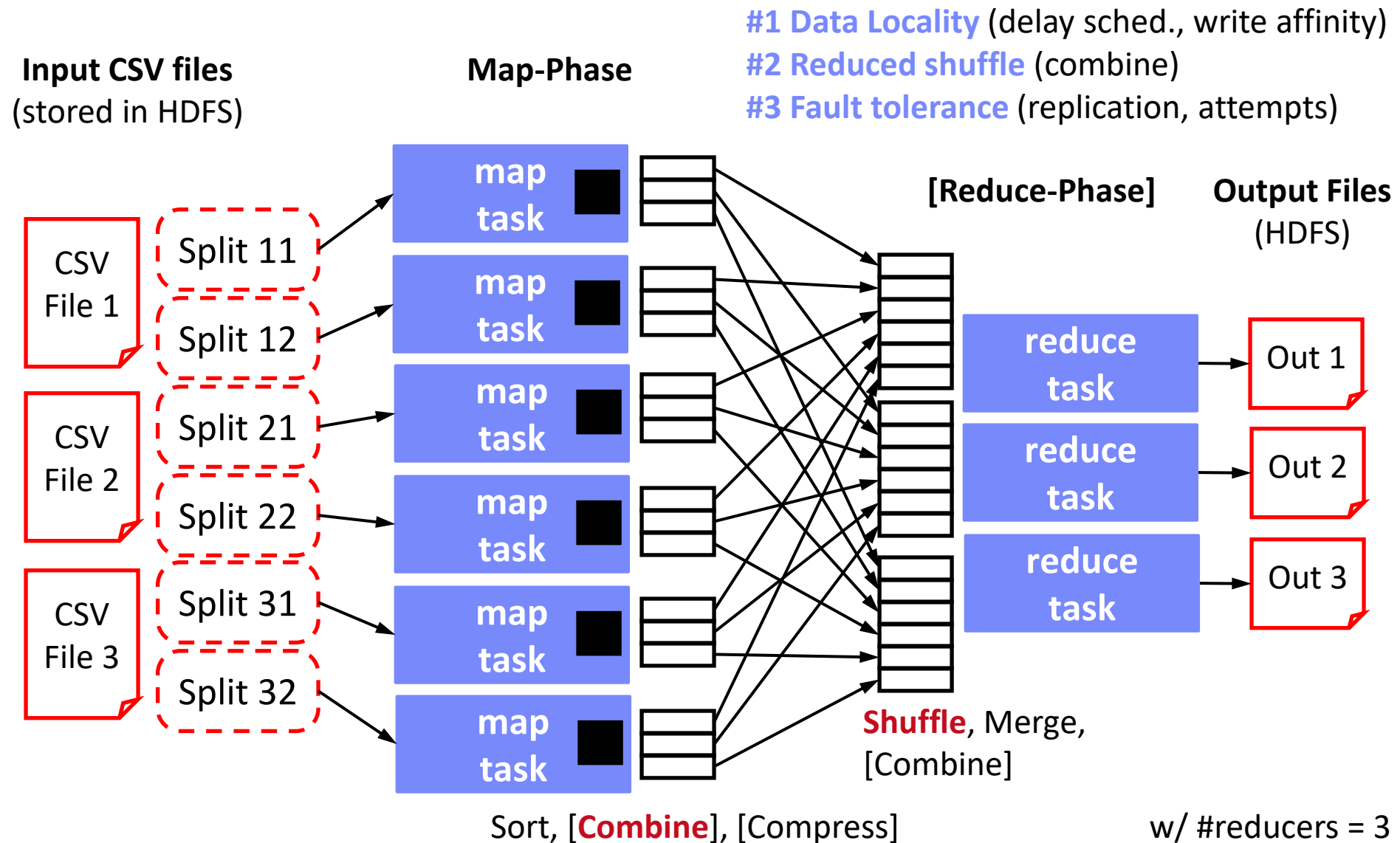
- **Horizontal parallelism** on data partitions
- Partitioning of inputs and intermediates (“support functions” and multiple queues)
- Process creation via **propagation tree** (fork tree)
- Partitioning: round-robin/range/hash

■ Example **Hash Partitioning**:

- For all $k \in R / k \in S$
- $\text{pid} = \text{hash}(k) \% n$



Excursus: MapReduce – Execution Model



Fine-grained Parallelism

[Viktor Leis, Peter A. Boncz, Alfons Kemper, Thomas Neumann: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. **SIGMOD 2014**]

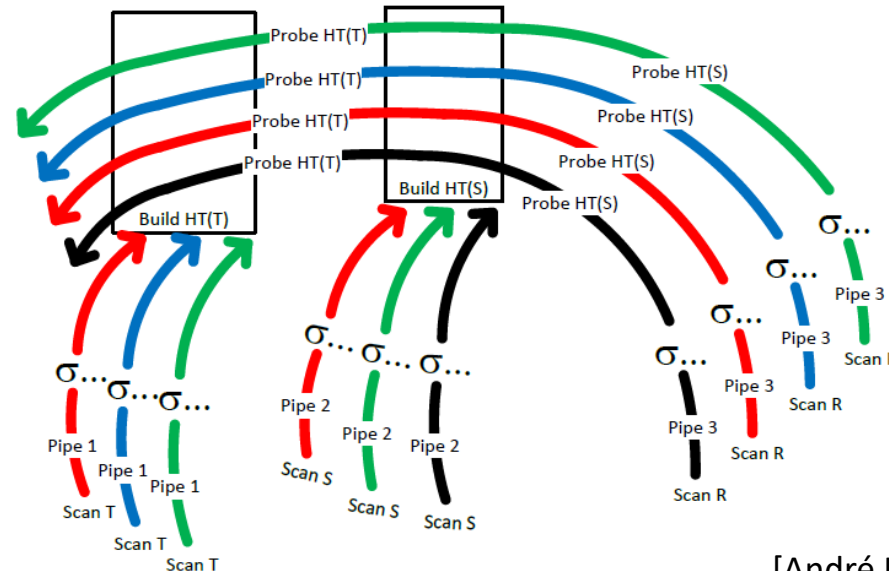


■ Motivation

- Non-uniform memory architecture (NUMA)
- **Load imbalance** / **serial fraction** due to plan-driven parallelism

■ Scheduler (dispatcher)

- Fixed number of workers to avoid over-provisioning
- Morsel: segment of tuples (e.g., 100K)
- Task: operator pipeline on morsel
- Task distribution at runtime w/ static partitioning + work stealing
- NUMA data locality



■ Hybrid Interpreted/compiled

- Exchange plans at morsel granularity

[André Kohn, Viktor Leis, Thomas Neumann: Adaptive Execution of Compiled Queries. **ICDE 2018**]



Fine-grained Parallelism, cont.

[Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, Weiwei Gong: A Morsel-Driven Query Execution Engine for **Heterogeneous Multi-Cores**. PVLDB 12(12) 2019]

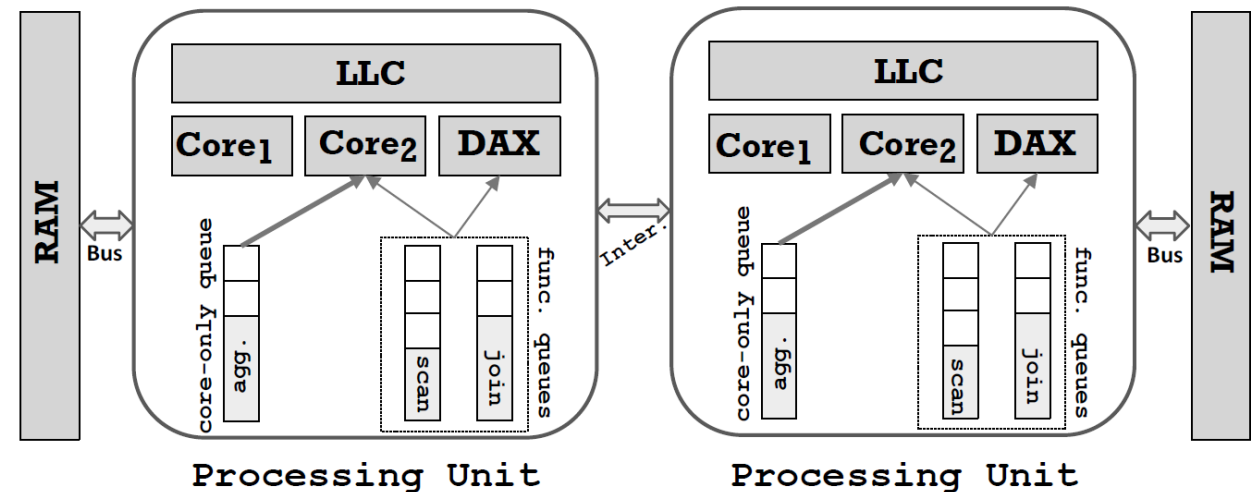


■ Motivation, cont.

- **Dark silicon** due to power and thermal constraints
- Sparc M7 platform w/ on-die ASIC, Data Analytics Accelerator (DAX)

■ Extensions

- Pipeline decomposition for **function-specific cores**
- Cost-based work submission to accelerator
- DAX: scan&filter, select, semi-join



■ Similar Abstractions for CPU/GPU balancing

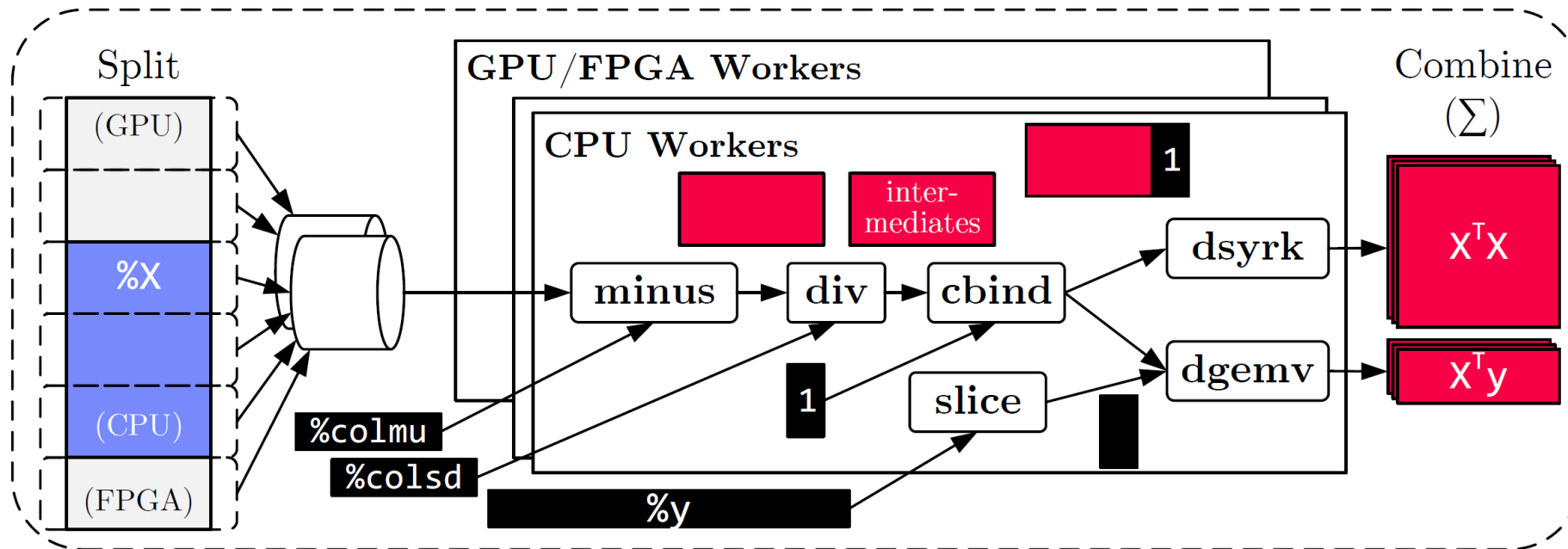
Excursus: DAPHNE Vectorized/Tiled Execution



■ Example

- Data placement on CPUs, GPUs, FPGAs
- Fused pipeline for `scale()` and `lmDS()`

```
(%9, %10) = fusedPipeline1(%X, %y, %colmu, %colsd) {
```



Workload Management

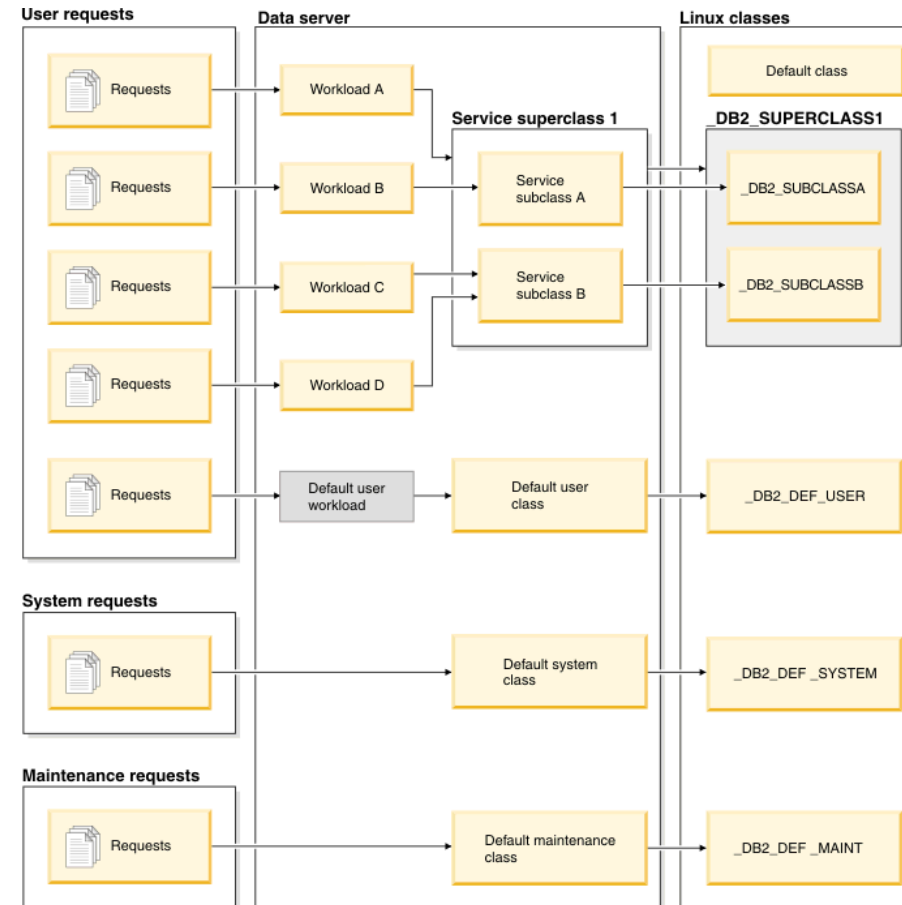


■ Example: DB2 Workload Management

- Concurrency thresholds for incoming requests
- Stop/continue/remap on violated thresholds
- Map DB2 service classes to Linux classes
- Linux cgroups (control groups) for resource isolation

[https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.wlm.doc/doc/c0053451.html]

[https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.wlm.doc/doc/c0053465.html]



Workload Management, cont.

- **Example H-Store → VoltDB**
 - Cluster of **single-threaded storage and execution engines**
 - No disk-based logging or locking

Result: No Multi-threading!!!

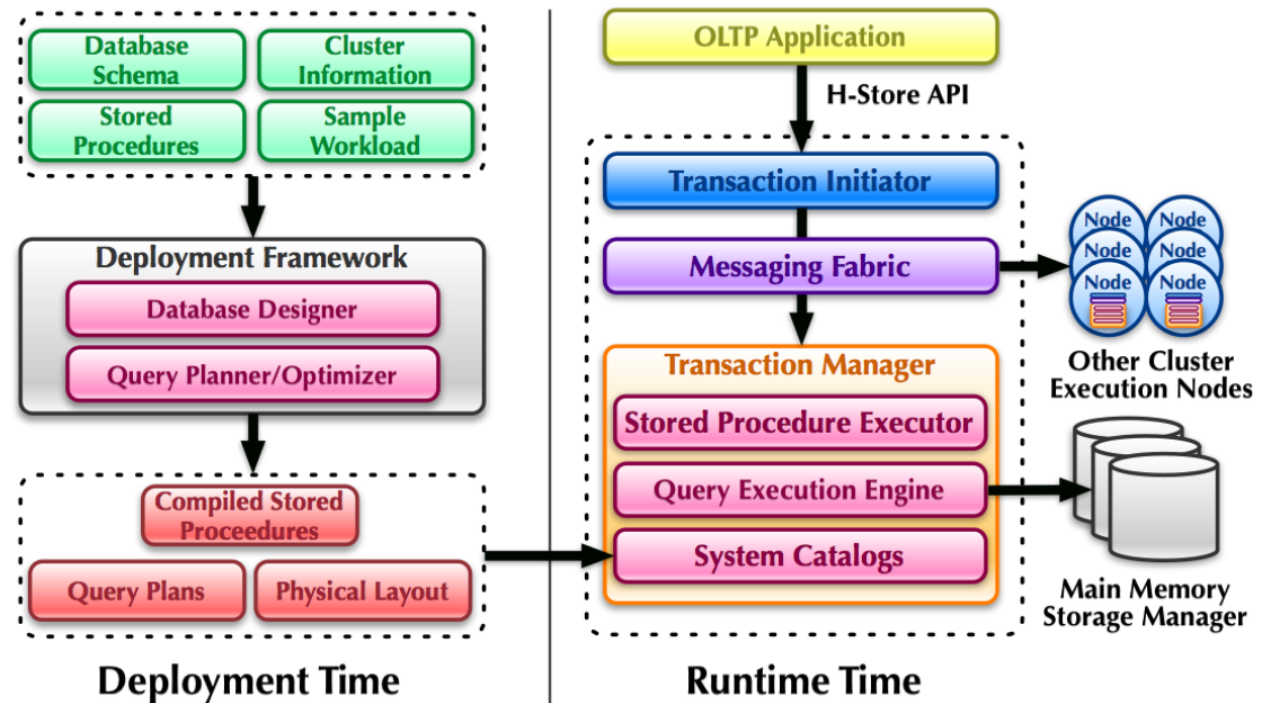
- ◆ Heaviest TPC-C Xact reads/writes 200 records
 - ◆ Less than 1 msec!!
- ◆ Run all commands to completion; single threaded
- ◆ Dramatically simplifies DBMS
 - ◆ No B-tree latch crabbing
 - ◆ No pool of file handles, buffers, threads, ..

Multiple cores can be handled by multiple logical sites per physical site



[Robert Kallman et al.: H-store: a high-performance, distributed main memory transaction processing system. **PVLDB 1(2) 2008**]

[Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, Pat Helland: The End of an Architectural Era (It's Time for a Complete Rewrite). **VLDB 2007**]



■ SAP HANA

- Main column store and delta CSB tree
- Thread pool for network clients
- **Scheduler for heavy-weight requests**
(single- or multi-task intra-query parallelism)

[Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, Kai-Uwe Sattler: Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. TPCTC 2014]



“[...] the default configuration of SAP HANA **favors analytical throughput** over transactional throughput”

■ UDFs via OpenMP

- OpenMP (since 1997, Open Multi-Processing)
- **DOALL parallel loops**
(independent iterations)
- **SAP HANA: custom OpenMP backend** for intercepting tasks
→ DB job scheduler (w/ priorities)

```
#pragma omp parallel for reduction(+: nnz)
for (int i = 0; i < N; i++) {
    int threadID = omp_get_thread_num();
    R[i] = foo(A[i], threadID);
    nnz += (R[i]!=0) ? 1 : 0;
}
```

[Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, Kai-Uwe Sattler: Extending database task schedulers for multi-threaded application code. **SSDBM 2015**]



Thanks

- Vectorization and SIMD
- Query Compilation
- Query Parallelization
- Next Lectures
 - 04 Background Query Optimization [May 26, Christina]
 - 05 Experiments and Reproducibility [Jun 09]
 - Project Submissions [Jul 08 EOD]
 - Project Presentations all Teams [Jul 14]