

# DEVVIEW 2014

DEVELOPERS  
CONFERENCE 2014



# Vectorized Processing in a Nutshell

DEVIEW  
2014

---

김형준  
Gruter

# Who I am

김형준, babokim@gmail.com

www.jaso.co.kr

클라우드컴퓨팅구현기술 등 저



Apache Tajo Committer

Gruter(BigData)

NHN(Hadoop)

SDS(공공 SI, 전자 ITO)



**Query Compilation**

**Vectorized** **LLVM**

**Cache Miss** **SIMD**

**CPU pipeline**

**Branch misprediction**

**이건 무슨 외계어?**

**SI 개발자가 왜 여기까지 왔나?**

# Hadoop 쉽다며? Hadoop 빠르다며?



# 그래서 다양한 시도가

SQL을 이용한 분석

메모리를 잘 활용해보자!

실행 계획을 잘 만들어 보자!

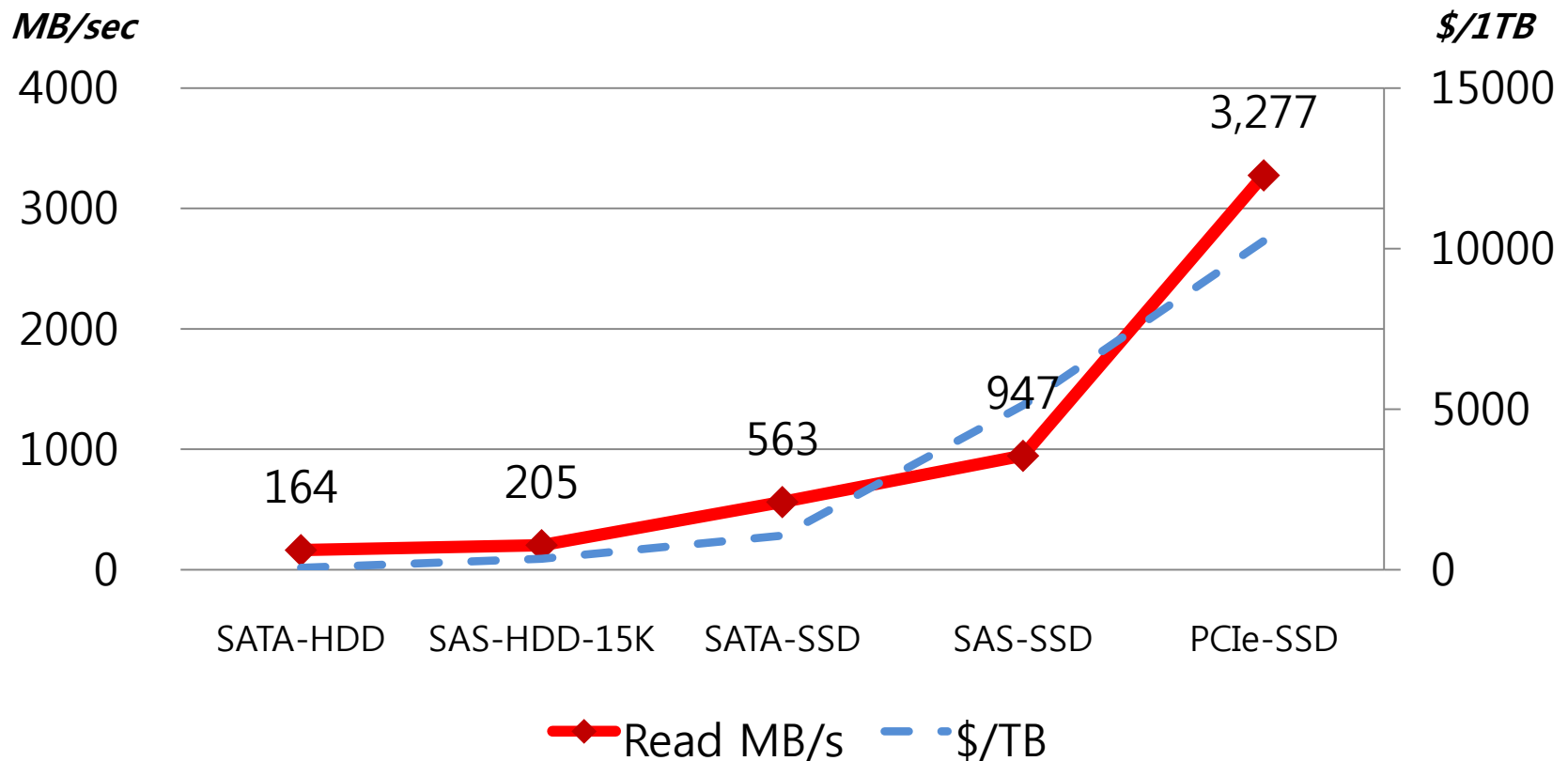
파일 포맷을 최적화 해보자!

...



# 단순하게 하드웨어로 해결?

처리해야 할 데이터가 많으니 Disk만 빠르면?



# 이렇게 했는데도 별로 안 빠르다!



**X 50대**

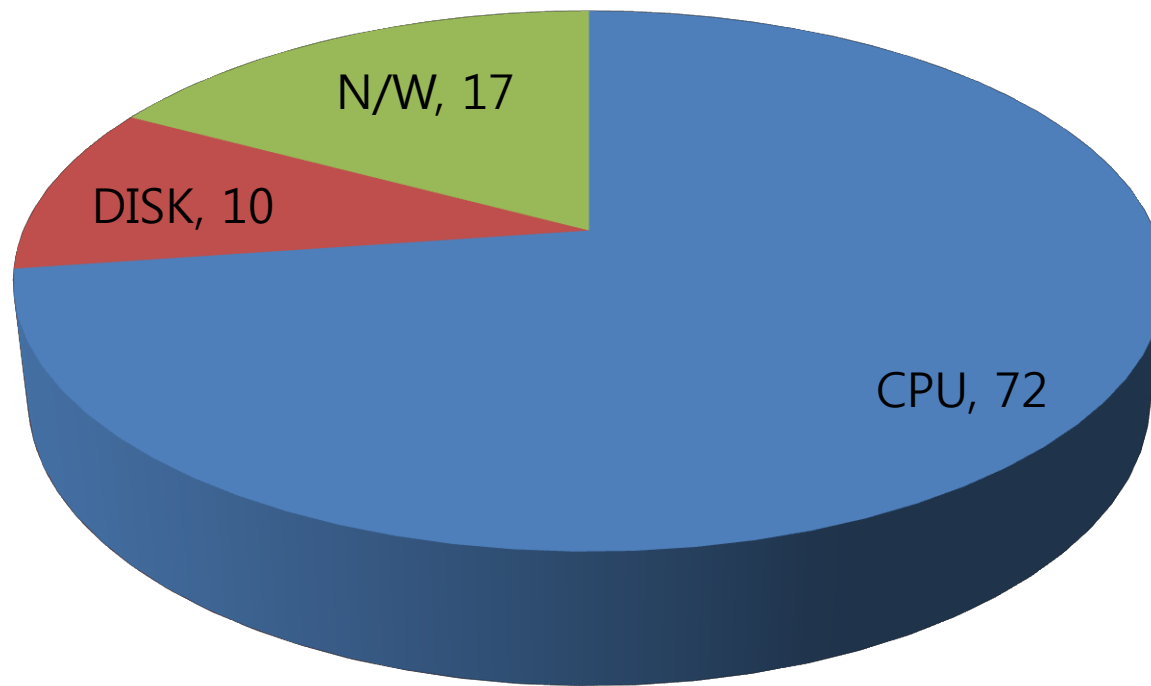
**240GB/s**

**10초에 2.4TB**

$\text{SAS} * 24\text{개} = 200\text{MB/sec} * 24 = 4.8\text{GB/s}$



# 데이터 분석 리소스 사용 유형



*HP-DL385P, SATA/HDD\*8, 10대  
Tajo, TPC-H Scale 1000질의*

# 그렇다고 계속 HDD만 사용?

더 빠른 CPU를 만들거나  
→ H/W 제조사가 할 일

CPU/Memory가 많은 장비를 사용  
(Scale-up)

→ 비용이 많이 들고  
메인 보드의 한계

# 우리(개발자)가 할 수 있는 것은

**CPU**를 마른 수건 짜듯이  
**잘 활용**

Vectorized

Query  
Compilation

# 어떻게 하면 CPU를 잘 활용할 수 있나?

한번에 더 많은 명령을 실행하게 하고

→ **CPU Pipeline**

CPU내 메모리를 잘 활용하고

→ **CPU Cache**

CPU에서 제공하는 기능을 잘 활용한다.

→ **SIMD**

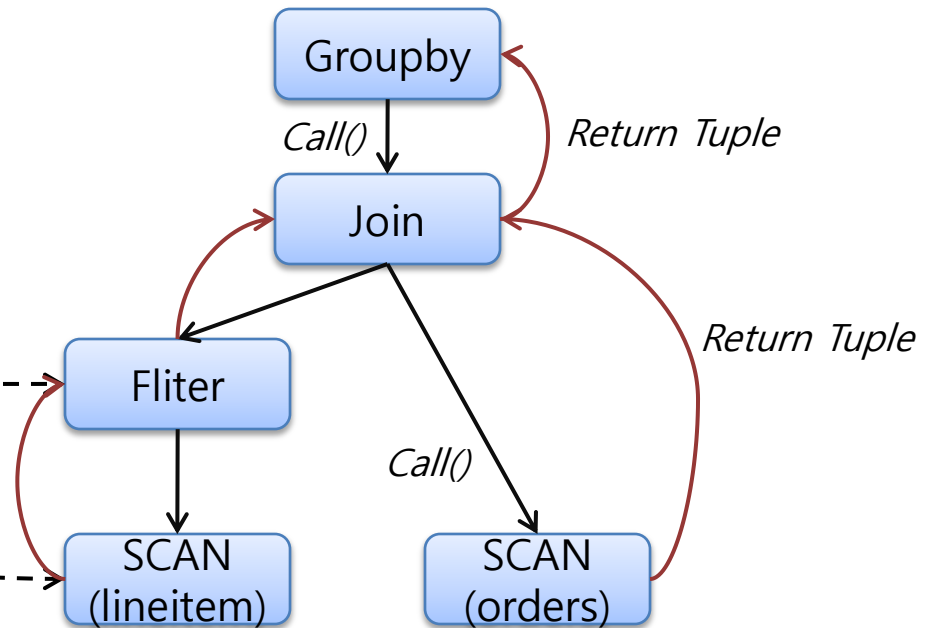
# DBMS는 어떻게 데이터를 처리?

## Basic Operator



```
SLECT o_custkey, l_lineitem, count(*)  
FROM lineitem a  
JOIN orders b ON a.l_orderkey = b.o_orderkey  
WHERE a.l_shipdate > '2014-09-01'  
GROUP BY o_custkey, l_lineitem
```

*Operator 조합으로  
Tree 형태의  
PhysicalPlan 생성*



# Tajo란?

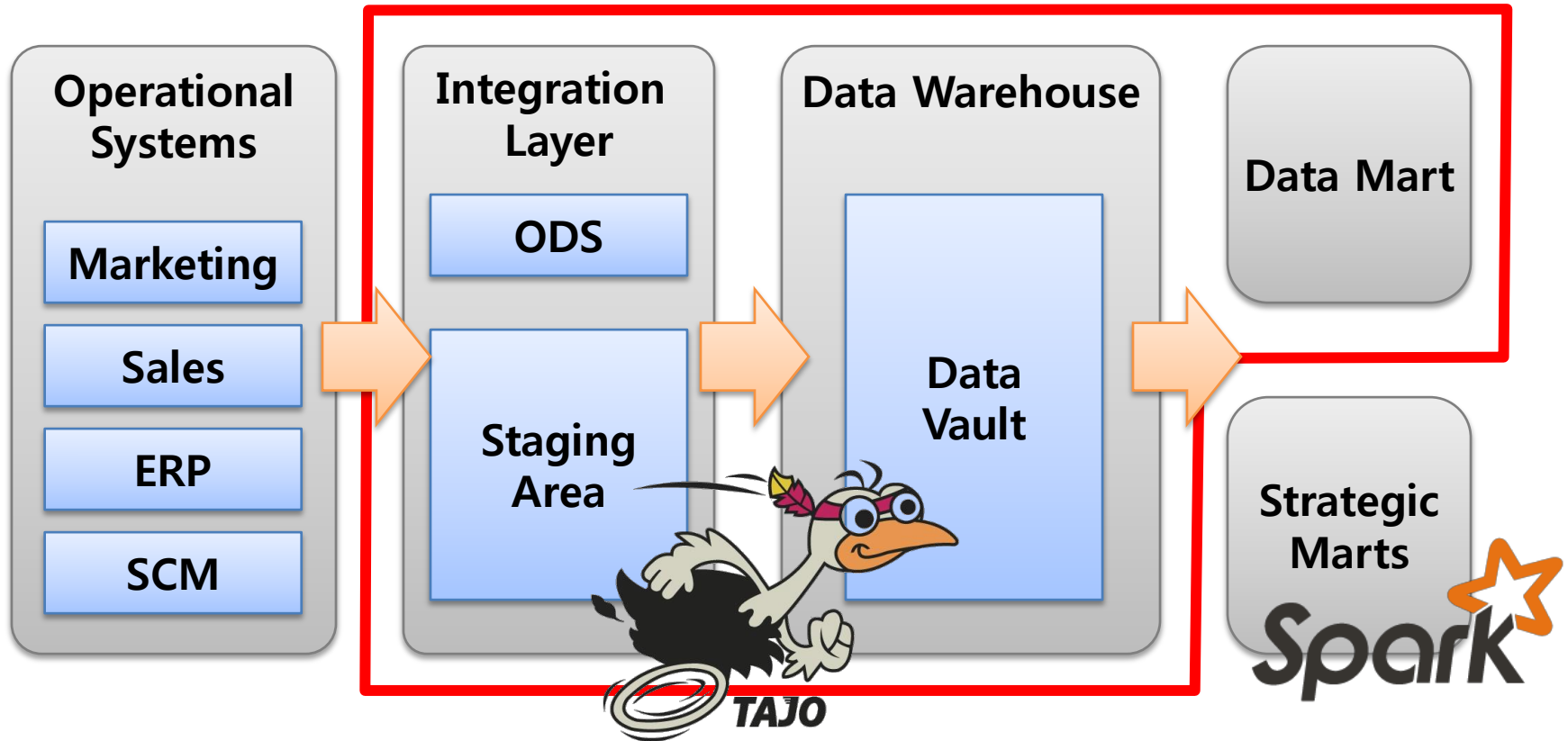


- SQL on Hadoop 솔루션으로 분류
  - Hadoop을 메인 저장소로 사용하는 Data Warehouse 시스템
  - 수초 미만의 Interactive 질의 + 수시간의 ETL 질의 모두 지원
- Apache Open Source (<http://tajo.apache.org>)
- 자체 분산 실행 엔진
  - MapReduce가 아닌 자체 분산 실행 엔진 이용
- 다양한 성능 최적화 기법 도입
  - Join Optimization, Dynamic Planning
  - Disk balanced scheduler, Multi-Level Count Distinct
  - Query Compilation, Vectorized Engine(2015 1Q) 등
- 이미 엔터프라이즈에 적용
  - 상용 DW Win back, 수TB/일 처리

[http://events.linuxfoundation.org/sites/events/files/slides/ApacheCon\\_Keuntae\\_Park\\_0.pdf](http://events.linuxfoundation.org/sites/events/files/slides/ApacheCon_Keuntae_Park_0.pdf)

<http://spark-summit.org/wp-content/uploads/2014/07/Big-Telco-Bigger-Real-time-Demands-Jungryong-Lee.pdf>

# Tajo DW 사례



상용 DW용 DBMS 역할 대체

**이제 본론으로...**



# CPU Pipeline

Instruction	1				2			
Fetch	Light Blue				Red			
Decode		Light Blue				Red		
Execute			Light Blue				Red	
Write				Light Blue				Red
Clock	1	2	3	4	5	6	7	8

Not Pipelined

Instruction	1	2			
Fetch	Light Blue	Red			
Decode		Light Blue	Red		
Execute			Light Blue	Red	
Write				Light Blue	Red
Clock	1	2	3	4	5

Pipelined

오/미/지: <http://www.digitalinternals.com/>

Instruction  
처리 속도 향상이 아닌  
처리 용량(throughput)  
향상

# CPU Pipeline의 어려움

## 다음과 같은 경우 Pipeline이 중지

- Structural Hazards
  - 여러 명령이 같은 하드웨어 리소스를 동시에 필요로 할 때
- Data Hazards
  - instruction들이 사용하는 data간 의존성 문제
  - ex)  $c = a + b;$   
 $e = c + d;$
- Control Hazards
  - 조건문 등이 있는 경우 다음 instruction을 예측하여 pipelining
  - 상황에 따라 분기 오예측 (branch misprediction)이 발생
  - 분기 예측이 잘못되었을 경우 pipelining에 있는 instruction이 모두 flush됨
  - Pipelining stage가 길수록 throughput이 떨어짐
  - 분기 오예측은 일반적으로 10-20 CPU cycle 소모

# Control Hazard 개선

## Loop unrolling

조건 비교

```
for (i = 0; i < 128; i++) {  
    a[i] = b[i] + c[i];  
}
```

Original Loop

```
for (i = 0; i < 128; i += 4) {  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
    a[i+2] = b[i+2] + c[i+2];  
    a[i+3] = b[i+3] + c[i+3];  
}
```

Unrolled Loop

## 컴파일러 옵션 또는 지시자로 설정 가능

지시자: #pragma GCC optimize ("unroll-loops")

컴파일 옵션: -funroll-loops 등

# Control Hazard 개선

## Branch avoidance

```
if (data[c] >= 128)
    sum += data[c];
```

```
2876      movsxd  rdx, DWORD PTR [rcx]
2877      cmp     edx, 128                      ; 00000080H
2878      jnl     SHORT $LN3@main ←
2879      add     rbx, rdx
2880 $LN3@main:
```

### Branch version

```
for (int i = 0; i < num; i++) {
    if (data[i] >= 128) {
        sum += data[i];
    }
}
```

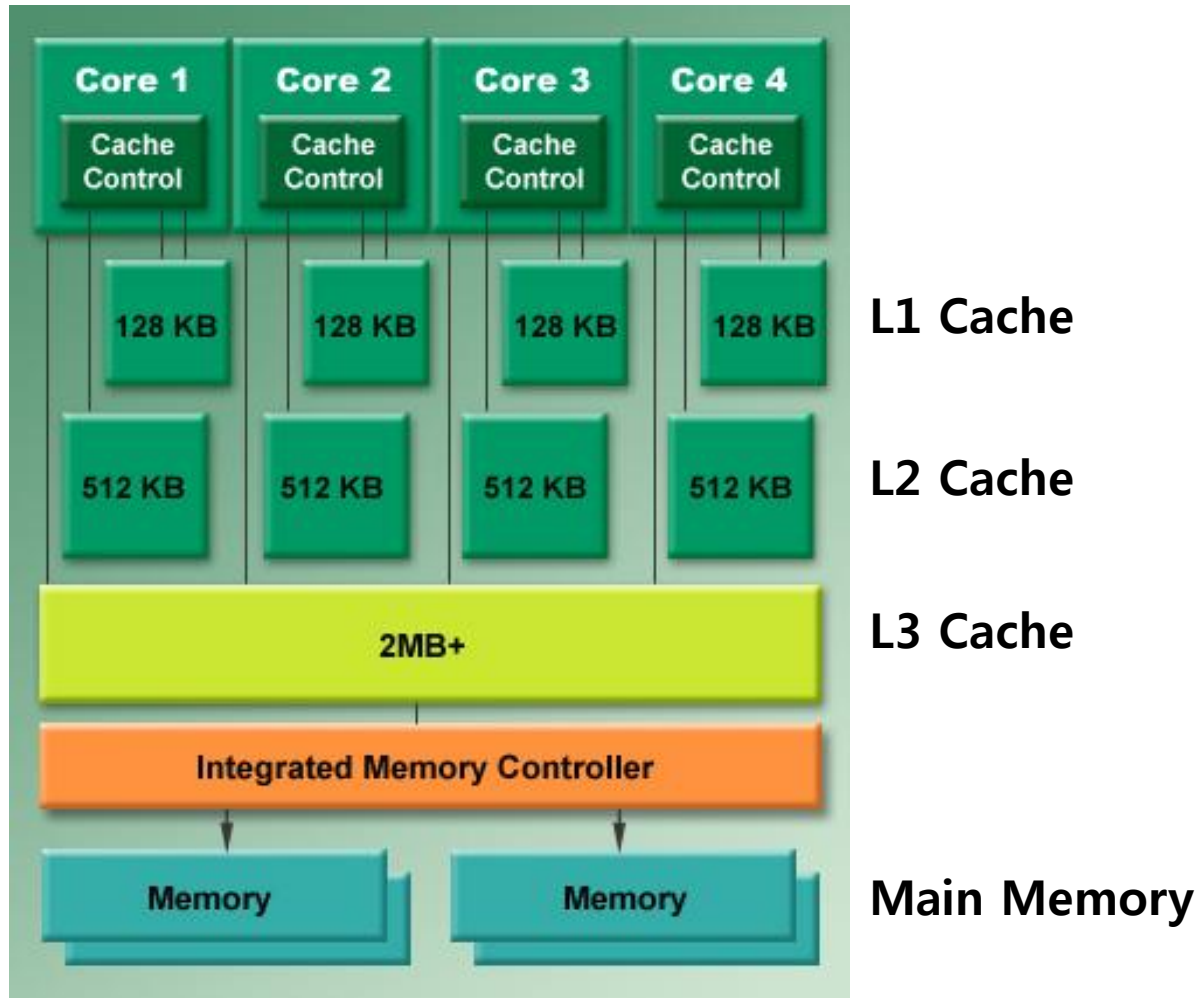
**Random data : 35,683 ms**  
**Sorted data : 11,977 ms**

### Non branch version

```
for (int i = 0; i < num; i++) {
    int v = (data[i] - 128) >> 31;
    sum += ~v & data[i];
}
```

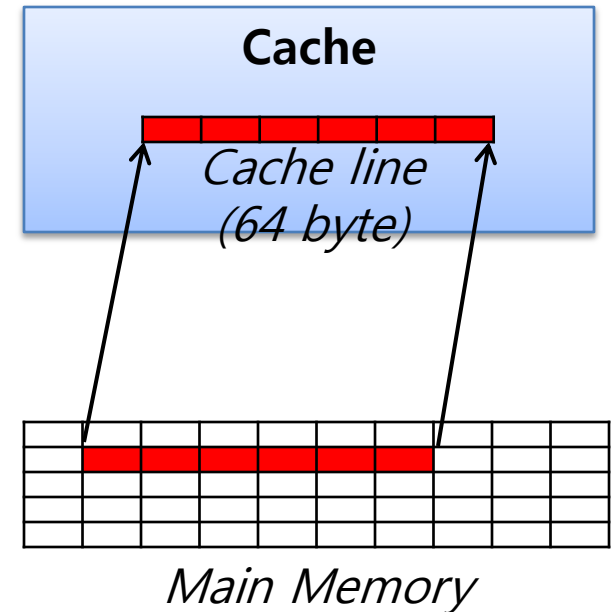
**11,977 ms**  
**10,402 ms**

# Memory Hierarchy



# CPU Cache 특징

- Spatial locality
  - cache line (64 bytes) 단위 인접한 데이터까지 메모리로부터 읽음
  - File System의 Block size와 같은 개념
- Temporal locality
  - cache의 크기는 제한적이며 캐쉬 유지 전략은 LRU
  - 즉, 가장 최근 사용된 것이 캐쉬에 남아 있음



# CPU Hit 비교

## Classic Example of Spatial Locality

### Case 1

```
for (i = 0 to size)
  for (j = 0 to size)
    do something with ary[j][i]
```

인접하지 않은 메모리를 반복해서  
읽기 때문에 비효율적

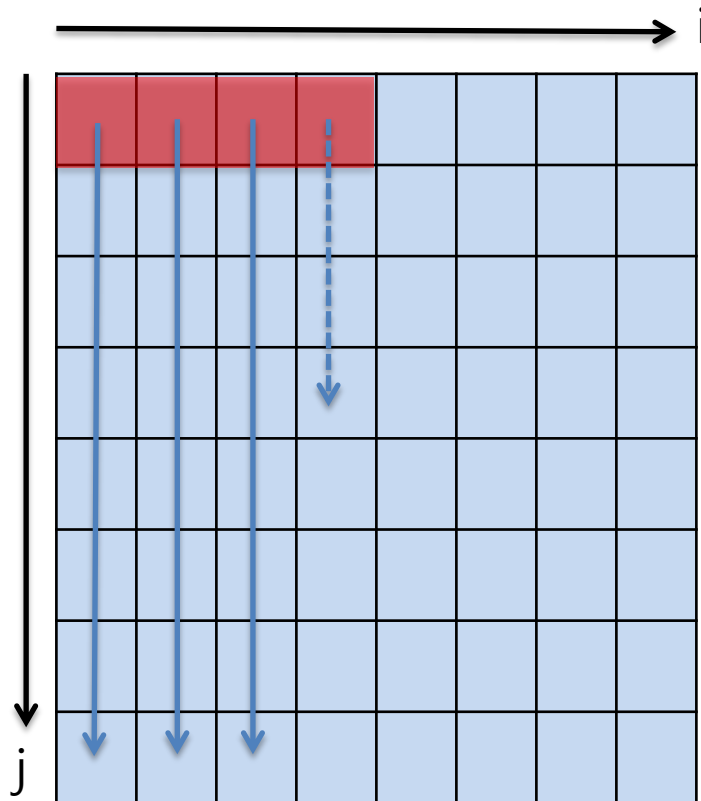
### Case 2

```
for (i = 0 to size)
  for (j = 0 to size)
    do something with ary[i][j]
```

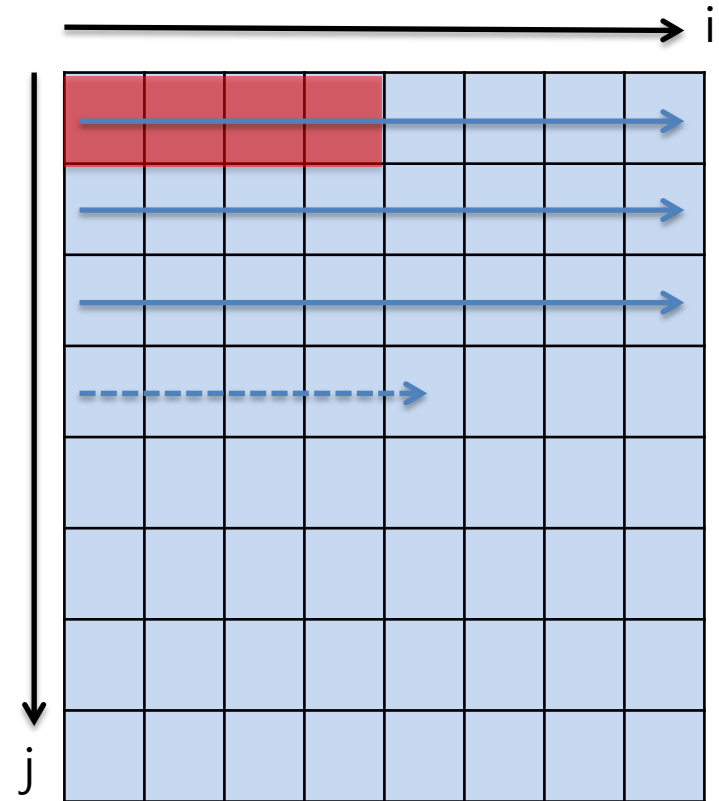
순차 읽기로 인한 높은 cache hit

# CPU Hit 비교

 Cache Line



Case 1



Case 2



# DBMS에서의 CPU Cache

- Tuple을 Row 단위로 저장할 경우
  - Row의 크기가 커서 Cache Hit율이 낮음
- Column 단위로 저장할 경우
  - Cache Hit율이 높음
  - 동일한 Operator를 동일 컬럼에서 여러 개의 데이터를 한번에 수행 가능(SIMD)
  - 하지만 기존의 처리 방식(Tuple-at-a time)을 사용하지 못함
    - 시스템 전반적인 변경이 필요

# SIMD

- Single Instruction Multiple Data
  - CPU에서 제공하는 병렬 처리 명령 Set
  - 사칙 연산 및 논리연산을 SIMD 명령으로 제공
- Pipeline은 명령을 병렬처리, SIMD는 **데이터를 병렬처리**
- Intel CPU의 SIMD는 MMX라는 이름으로 제안, SSE, AVX로 진화 중
- Nehalem은 단일 명령으로 128bit 데이터
- Sandy Bridge 이후부터는 256bit 데이터에 대한 명령 셋 제공
  - 256 bit의 크기 = 16 Shorts, 8 Integers/Floats, 4 Long/Double

# SIMD

Scalar

x

\*

y

vs.

---

$x * y$

SIMD

X3 X2 X1 X0

\*

\*

\*

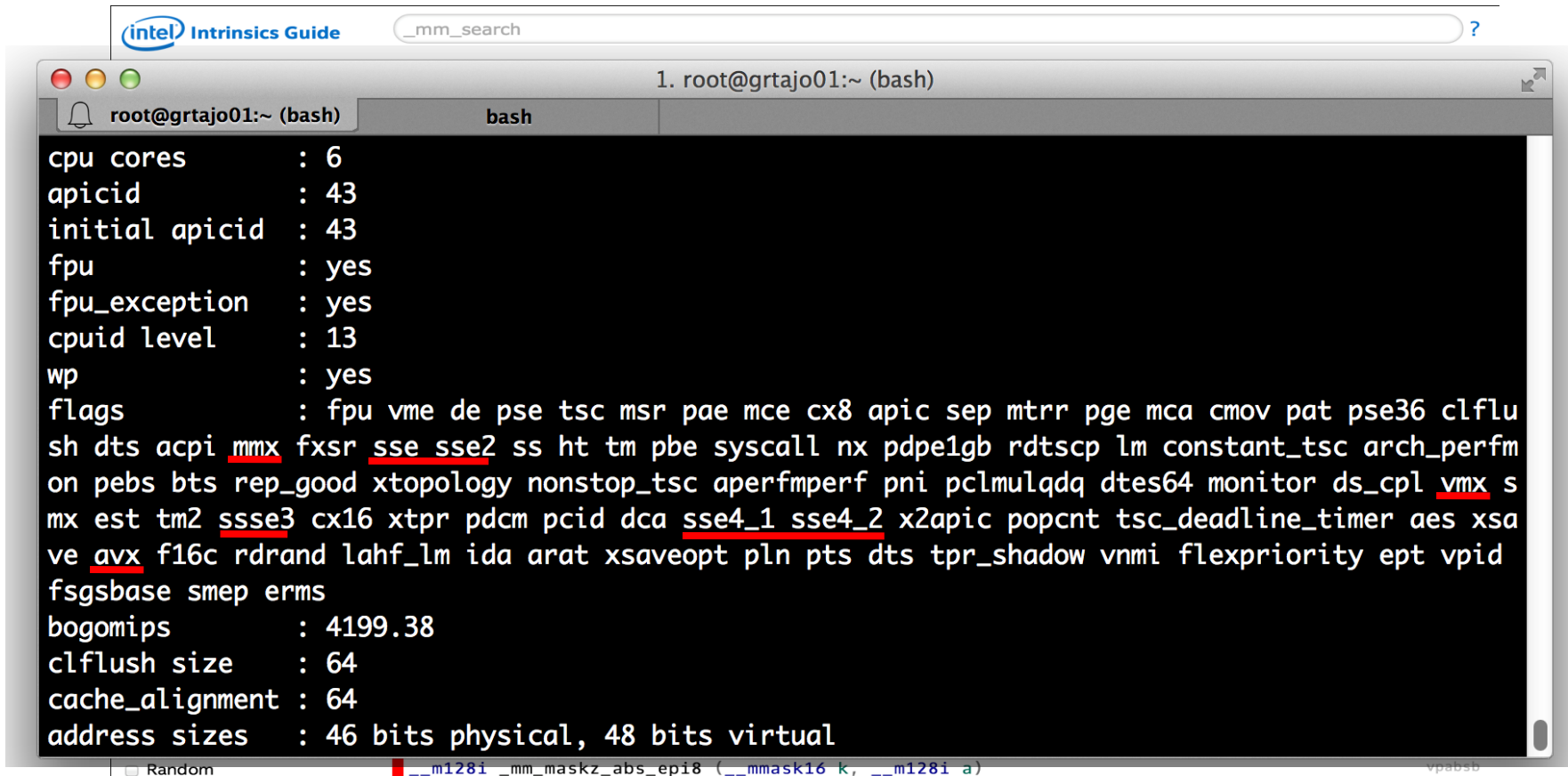
\*

Y3 Y2 Y1 Y0

---

$X3 * Y3$   $X2 * Y2$   $X1 * Y1$   $X0 * Y0$

# Intel SIMD Instruction



The screenshot shows a terminal window with a title bar that includes the Intel Intrinsic Guide logo and a search bar. The terminal output lists various system parameters and SIMD instructions. The instructions listed are: `mmx`, `sse`, `sse2`, `ss`, `ht`, `tm`, `pbe`, `syscall`, `nx`, `pdpe1gb`, `rdtscp`, `lm`, `constant_tsc`, `arch_perfmon`, `pebs`, `bts`, `rep_good`, `xtopology`, `nonstop_tsc`, `aperfperf`, `pni`, `pclmulqdq`, `dtes64`, `monitor`, `ds_cpl`, `vmx`, `smx`, `est`, `tm2`, `ssse3`, `cx16`, `xtpr`, `pdc`, `pcid`, `dca`, `sse4_1`, `sse4_2`, `x2apic`, `popcnt`, `tsc_deadline_timer`, `aes`, `xsave`, `avx`, `f16c`, `rdrand`, `lahf_lm`, `ida`, `arat`, `xsaveopt`, `pln`, `pts`, `dt`, `tpr_shadow`, `vmi`, `flexpriority`, `ept`, `vpid`, `fsgsbase`, `smep`, and `erms`. The instructions `mmx`, `sse`, `sse2`, `ssse3`, `sse4_1`, `sse4_2`, `avx`, and `vmx` are underlined in the original image. The terminal also shows system information such as 6 CPU cores, 43 APIC IDs, and 46 bits physical / 48 bits virtual address sizes.

```
cpu cores      : 6
apicid        : 43
initial apicid : 43
fpu           : yes
fpu_exception : yes
cpuid level   : 13
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflu
sh dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
on pebs bts rep_good xtopology nonstop_tsc aperfperf pni pclmulqdq dtes64 monitor ds_cpl vmx s
mx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm ida arat xsaveopt pln pts dt tpr_shadow vmi flexpriority ept vpid
fsgsbase smep erms
bogomips      : 4199.38
clflush size  : 64
cache_alignm  : 64
address sizes : 46 bits physical, 48 bits virtual
```

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

# DBMS 에서는

- Tuple Memory Model
  - Tuple을 메모리에 저장하는 방식
  - Cache hit율에 영향
- Tuple Processing Model
  - 처리하는 Tuple의 단위
  - Cache hit, CPU Pipeline, SIMD 활용에 모두 영향

# Tuple Memory Model

**Relation**

Id	Name	Age
101	Alice	22
102	Ivan	37
104	Peggy	45
105	Victor	25
108	Eve	19
109	Walter	31
112	Trudy	27
113	Bob	29
114	Zoe	42
115	Charlie	35

**NSM representation**

age 1

101	Alice	22	102
Ivan	37	104	Peggy
	45	105	Victor
25	108	Eve	19

age 2

109	Walter	31	112
Trudy	27	113	Bob
	29	114	Zoe
42	115	Charlie	35

OR

**DSM representation**

Id	Name	Age
101	Alice	22
102	Ivan	37
104	Peggy	45
105	Victor	25
108	Eve	19
109	Walter	31
112	Trudy	27
113	Bob	29
114	Zoe	42
115	Charlie	35

*N-array Storage Model*

*Decomposed Storage Model*

# Tuple Memory Model

- Row-store (NSM)
  - 하나의 row를 레코드 단위로 하여 물리적으로 연속적인 바이트 열로 저장
  - 장점: 레코드 단위의 쉬운 삽입/삭제/업데이트 가능
  - 단점: I/O의 최소 단위는 레코드
  - OLTP에 적합
- Column-store (DSM에서 발전)
  - 컬럼을 별도의 파일에 저장
  - 장점
    - 필요한 컬럼들에 대해 컬럼 단위 I/O 가능
    - 매우 빠른 읽기 성능
    - 더 높은 압축률
  - 단점
    - 많은 컬럼을 읽을 경우 tuple 형태로 만드는 비용이 큼
    - 파일 쓸 때 여러 파일에 대해 I/O가 발생
    - 읽기 위주의 분석적 처리에 적합

# **Tuple Processing Model**

**Tuple-at-a-time**

**Block-at-a-time**

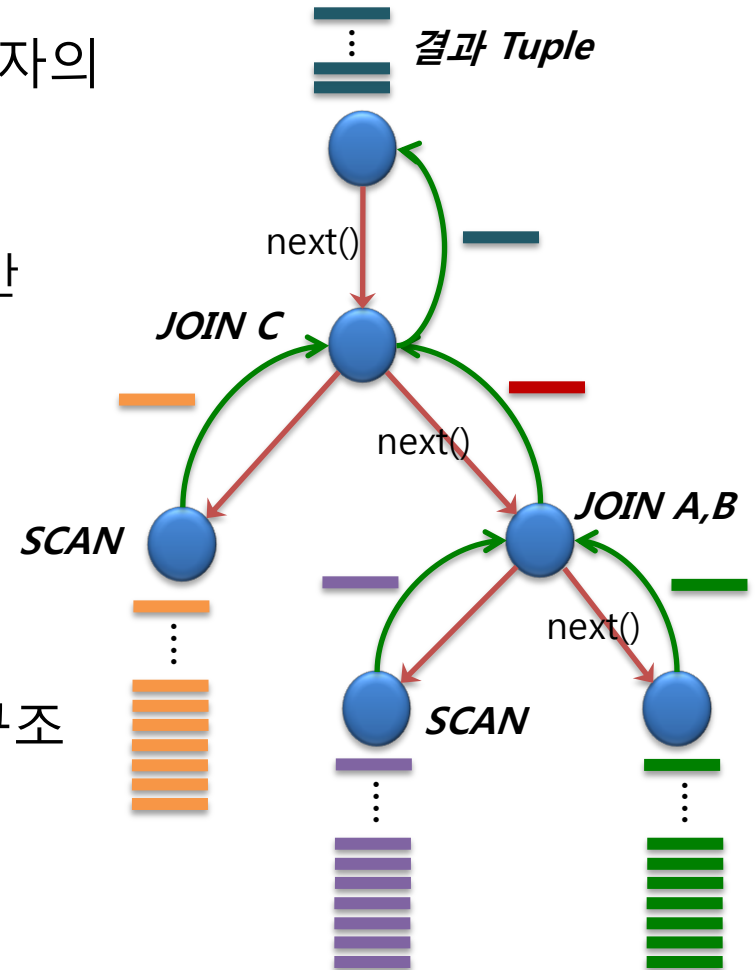
**Column-at-a-time**

**Vectorized**



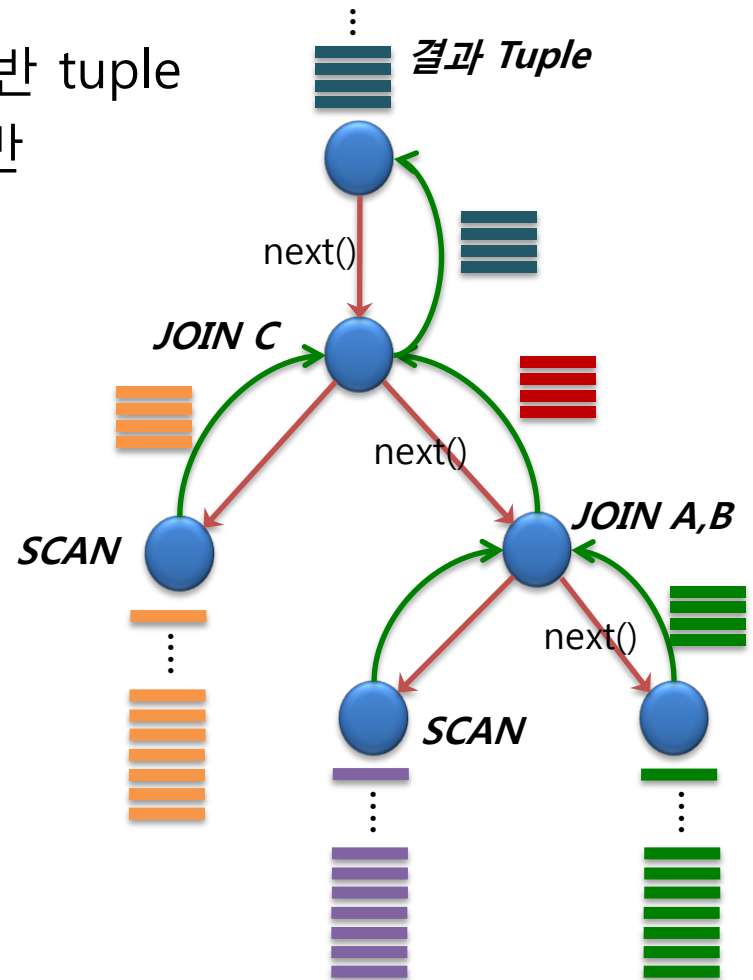
# Tuple-at-a-time Model

- next() 호출마다 재귀적으로 하위 연산자의 next() 호출
- next()의 반환 값은 하나의 Tuple
- N-array Storage Model (NSM) 레코드 기반
- CPU 활용 측면에서 비효율적
  - 함수 호출 비용 (20 CPU cycle)
  - Pipeline 활용도 낮음
  - SIMD 활용도 낮음
  - 빈번한 instruction/data cache miss
  - NSM으로 인한 cache miss
- MySQL, Pig, Tajo(AS-IS)가 기반하는 구조



# Block-at-a-time Model

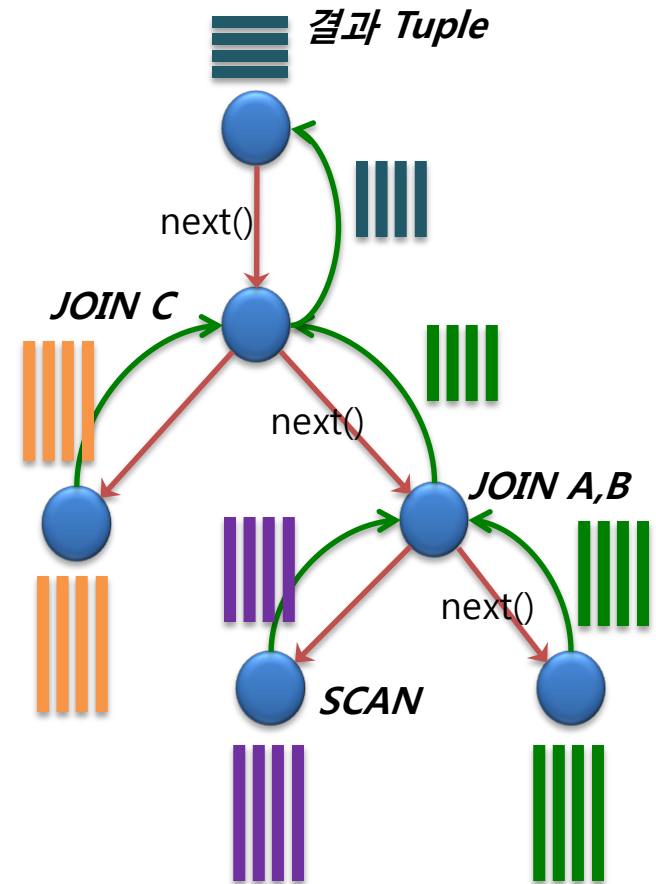
- 한번의 next() 호출에 N개의 NSM 기반 tuple
- N-array Storage Model (NSM)에 기반
- Impala, PostgreSQL 등
- 장점
  - 함수 호출 비용 감소
  - instruction/data cache hit 향상
- 단점
  - Pipelining 활용도 여전히 낮음
  - SIMD 활용도 낮음
  - NSM으로 인한 cache miss



*Block oriented processing of Relational Database operations in modern Computer Architectures*

# Column-at-a-time Model

- Decomposed Storage Model 사용
- 각 컬럼 배열 자료 구조에는 전체 Row의 데이터를 반환
- MonetDB에서 사용
- 장점
  - 함수 호출 비용 감소
  - Instruction/data cache hit 향상
  - Pipelining 활용도 높음
  - SIMD 활용도 높음
- 단점
  - 중간 데이터 유지 비용 높음
  - 여전히 높은 cache miss 빈도
    - 두 컬럼의 데이터를 이용하는 연산의 경우



*Monet: a next-Generation DBMS Kernel*

# Column-at-a-time Model

- 성능 비교
  - MySQL: 26.2 sec, MonetDB: 3.7 sec
  - TPC-H-Q1(1 Table Groupby, Orderby, 결과 데이터는 4건)
  - In-memory 데이터에 대한 비교 실험 결과
- 논리연산/사칙 연산 별로 별도 Primitive 연산자 필요

```
/* Returns a vector of row IDs' satisfying comparison  
 * and count of entries in vector. */
```

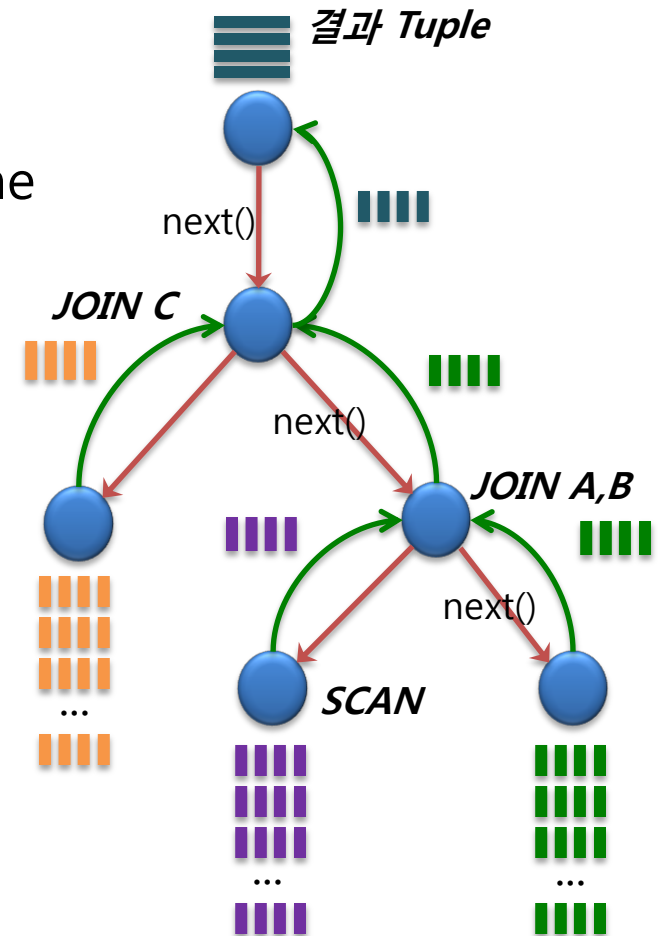
```
int select_gt_float(int* res, float* column, float val, int n) {  
    int idx = 0;  
    for (int i = 0; i < n; i++) {  
        idx += column[i] > val;  
        res[idx] = j;  
    }  
    return idx;  
}
```

Greater-than primitive 연산자의 예

기본 데이터 타입, 사칙연산, 논리연산에 약 5천여개 필요

# Vectorized Model

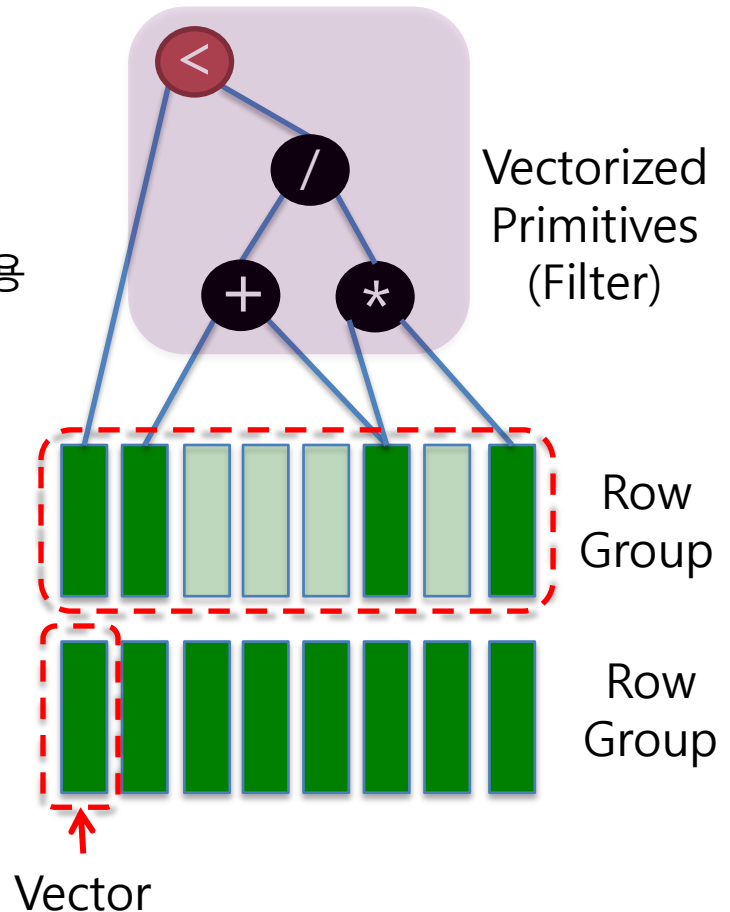
- 1개의 Column에 대해 Vector에 여러개의 데이터를 저장
- 서로 다른 Column vector를 CPU L2 Cache 크기(256KB)에 맞추어 row group 단위로 연산
- 조건식/계산식은 vector 단위로 처리
- 장점
  - 함수 호출 비용 감소
  - Instruction/data cache hit 향상
  - Pipelining 활용도 높음
  - SIMD 활용도 높음
  - Interpret overhead 최소화
- 단점
  - 새로운 모델로 개발 필요



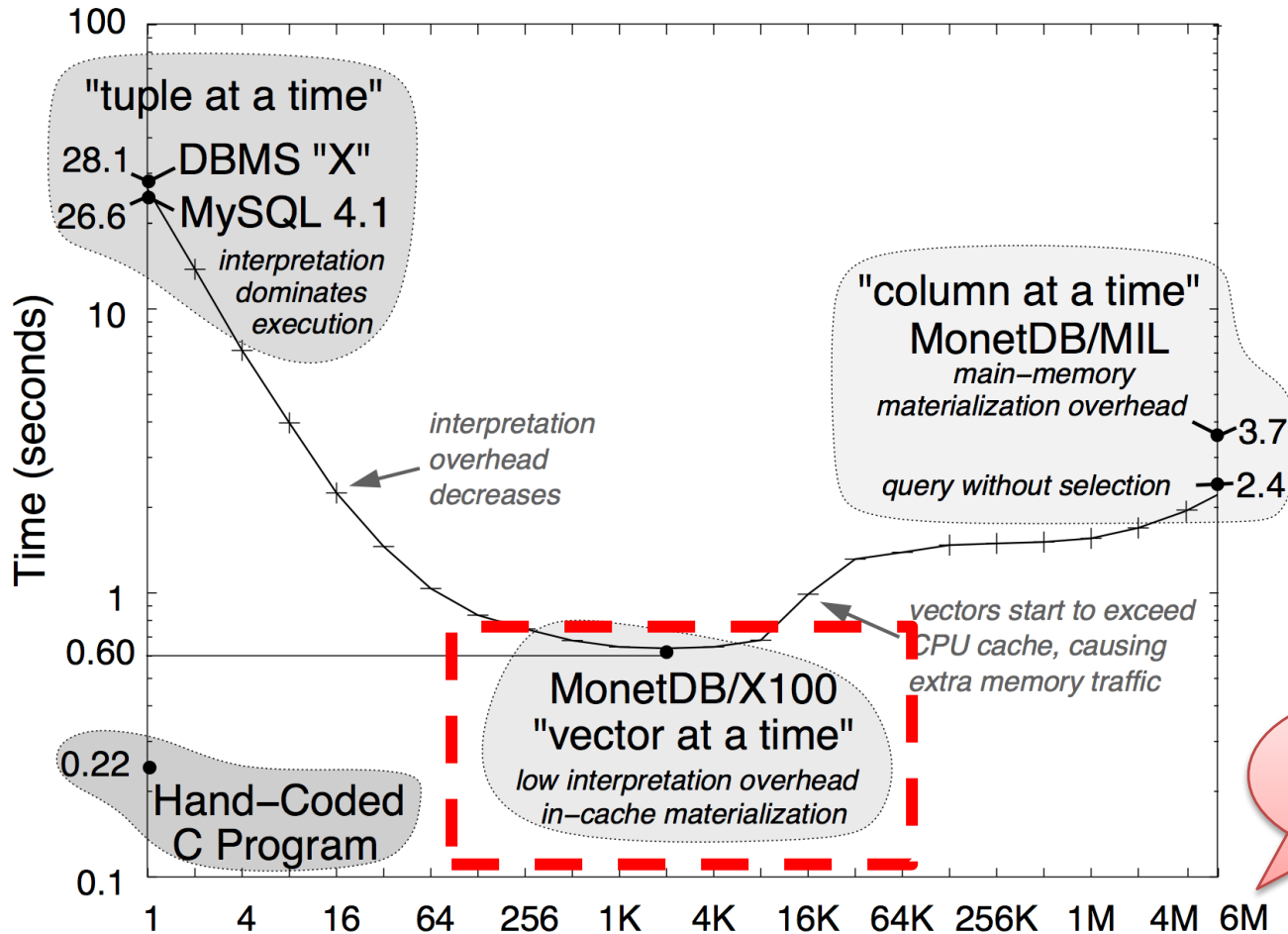
# Vectorized Model

- 하나의 operator 처리를 위해서 row group에 속하는 다수의 컬럼들을 액세스 필요
  - Column-at-a-time 방식은 그 과정에서 cache miss가 발생
  - Vectorized 방식은 각 row group의 vector들이 CPU L2 cache 사이즈에 맞도록 설계
  - CPU Pipelining, SIMD, CPU Cache 세 가지 활용을 극대화
- 
- MySQL(Tuple-at-a-time) : 26.2 secs
  - MonetDB(Column-at-a-time) : 3.7 secs
  - MonetDB X100(Vectorized) : 0.6 secs

## Filter 연산 예제



# Vectorized Model



Vector  
Size



# We always welcome new contributors!

General

<http://tajo.apache.org>

Issue Tracker

<https://issues.apache.org/jira/browse/TAJO>

Join the mailing list

[dev-subscribe@tajo.apache.org](mailto:dev-subscribe@tajo.apache.org), [issues-subscribe@tajo.apache.org](mailto:issues-subscribe@tajo.apache.org)

한국 User group

<https://groups.google.com/forum/?hl=ko#!forum/tajo-user-kr>



# Q&A



THANK YOU

