CMSC724: Modern Hardware

Amol Deshpande

University of Maryland, College Park

May 8, 2013

Outline

Motivation

- DBMSs built for 70's-80's hardware
- Current hardware is much much different
  - Need to rethink the design
- Key issues:
  - Pipelining $\rightarrow$ dependent code, branches bad
  - Multi-core
  - Caches
  - GPUs: lots of processing power, not clear how to use it
  - Increasingly NUMA architectures
  - FPGAs

Overview of "Modern" CPUs

- Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005
- Heavy use of instruction pipelining
  - Split a CPU instruction into large number of stages
    - 1993 Pentium: 5-stage pipeline, 2004 Penitum4: 31 pipeline stages
    - Example stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, etc...
    - More stages –> simpler architecture
    - More stages necessitates speculative execution
    - More stages –> Wasted work because of dependent instructions and branch misprediction

Overview of "Modern" CPUs

- Heavy use of instruction pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Overview of "Modern" CPUs

- Heavy use of instruction pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- Super-scalar architectures
  - Large number of independent pipelines
  - Hard to keep feeding data into them in many cases

Overview of "Modern" CPUs

- Heavy use of instruction pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- Super-scalar architectures
  - Large number of independent pipelines
  - Hard to keep feeding data into them in many cases

Overview of "Modern" CPUs

- Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005

int sel_lt_int_col_int_val(int n, int* res, int* in, int V) {

```
  for(int i=0,j=0; i<n; i++){
    /* branch version */
   if (src[i] < V)
      out[j++] = i;
    /* predicated version */
    bool b = (src[i] < V);
    out[j] = i;
    j += b;
  }
  return j;
}
```

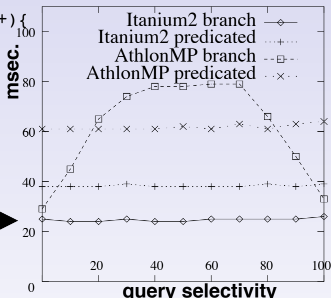
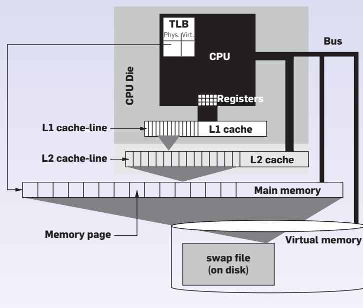


Figure 2: Itanium Hardware Predication Eliminates Branch Mispredictions

Overview of "Modern" CPUs

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- Memory Hierarchy



- On-chip caches (SRAM, vs DRAM for main memory)
  - Increasingly overall execution CPU bound, rather than I/O bound
  - Cache misses quite critical

Memory Hierarchy

- Issues:
  - DRAM latencies: 1-2 cycles in 80's, 300 cycles today (i.e., 2009)
  - L1 cache: usually split between instructions and data; L2, L3: unified
  - Capacity = 30KB-4MB; Line size = 64 bytes
  - Associativity: higher associative ==> better performance, but much higher cost (i.e., silicon cost)
    - Compulsory cache misses vs conflict misses
  - Translation lookaside buffer (TLB) (Note: paper called it "transition" – wrong)
    - Mapping between virtual addresses and real addresses
    - Double penalty if you get a TLB miss

Overview of "Modern" CPUs

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- Key innovations that help:
  - Vertical storage: Better utilization of cache lines
  - Bulk query algebra: Simplified algebra
  - Cache-conscious algorithms: Re-implementations to minimize cache misses
  - Memory access cost modeling
- Aside: always keep in mind Amdahl's law
  - In many of these cases, the maximum benefits are limited
  - Often not orders of magnitude

Outline

DBMSs on Modern Processors: Where does the time go?

- VLDB 1999
- Detailed study comparing multiple commercial DBMSs
  - Couldn't name the DBMSs because of the "DeWitt" clause

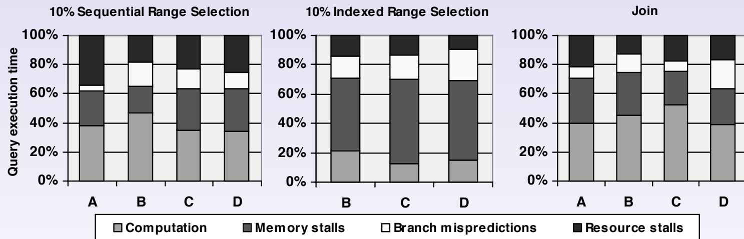


Figure 5.1: *Query execution time breakdown into the four time components.*

DBMSs on Modern Processors: Where does the time go?

- VLDB 1999

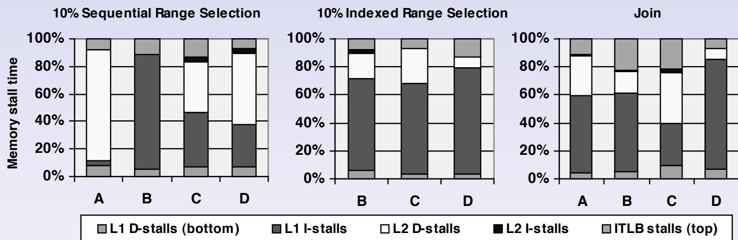


Figure 5.2: *Contributions of the five memory components to the memory stall time (T_M)*

DBMSs on Modern Processors: Where does the time go?

- VLDB 1999

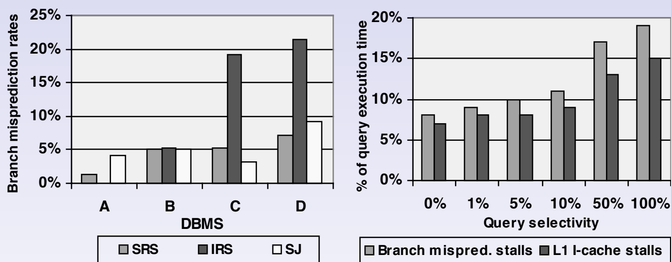


Figure 5.4: *Left: Branch misprediction rates. SRS: sequential selection, IRS: indexed selection, SJ: join. Right: System D running a sequential selection.* T_B *and* T_{L1I} *both increase as a function of an increase in the selectivity.*

MonetDB/X100

- Discussion from: "MonetDB/X100: Hyper-Pipelining Query Execution"; CIDR 2005
- Detailed experiments comparing where time went in a relational DBMS
- Even simple queries on MySQL: most time is not spent doing useful work
  - 62% spread over functions dealing with MySQL's record representation
- MonetDB (old version) suffered from memory bandwidth limitations
- X100: New vectorized query processor

Outline

Volcano iterator model

- Discussion from notes by Jens Teubner (ETH)
- Data passed from operator to operator using *next()*
- Problems:
  - Operators tightly interleaved $\rightarrow$ instruction cache misses
  - Large function call overhead
  - Combined state too large to fit into caches (data cache misses)
  - Single-tuple functions hard to optimize by compiler

MonetDB Column-at-a-time processing

- Discussion from notes by Jens Teubner (ETH)
- Operators consume and produce full columns
- Each sub-result fully materialized
- No pipelining
- Example:
  - sel-age := people-age.select(30, nil)
  - sel-id := sel-age.mirror().join(people-age)
  - tmp := [-] (sel-age, 30)

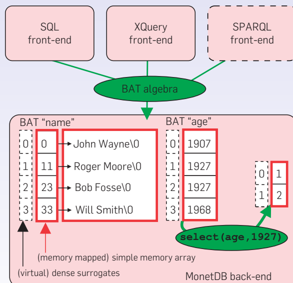
MonetDB Column-at-a-time processing

- Discussion from notes by Jens Teubner (ETH)
- Operators consume and produce full columns
- Each sub-result fully materialized
- No pipelining
- Example:
  - sel-age := people-age.select(30, nil)
  - sel-id := sel-age.mirror().join(people-age)
  - tmp := [-] (sel-age, 30)
- Advantages:
  - Tight loops conveniently fit into instruction caches,
  - Can be optimized ectively by modern compilers,
    - loop unrolling
    - vectorization (use of SIMD instructions)
  - Function calls are now out of the critical code path.

MonetDB/X100

- Still not ideal
  - Data may not fit in cache
  - Problems when intermediate results don't fit in main memory
- MonetDB aims for a middle ground
  - Still use iterator model, but pass vectors of tuples around
  - Large enough that the overheads are amortized

Figure 2: MonetDB: a BAT algebra machine.



Outline

Cache-conscious algorithms - Selections

- Conjunctive selection conditions; Ross; PODS 2001
- Need to apply k predicates, and find all tuples that satisfy all of them
  - Predicates are: f_1, ..., f_k
  - $r1[i]$ = the i'th attribute of r_1.

```
/* Algorithm Branching-And */
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) && ... && fk(rk[i]))
    {answer[j++] = i;}
  }

/* Algorithm Logical-And */
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) & ... & fk(rk[i]))
    {answer[j++] = i;}
  }

/* Algorithm No-Branch */
for(i=0;i<number_of_records;i++) {
  answer[j] = i;
  j += (f1(r1[i]) & ... & fk(rk[i]));
  }
```

Cache-conscious algorithms - Selections

- Conjunctive selection conditions; Ross; PODS 2001
- Need to apply k predicates, and find all tuples that satisfy all of them
  - Predicates are: f_1, ..., f_k
  - $r1[i]$ = the i'th attribute of r_1.
- Can't expect an optimizer to transform the three plans into each other
  - Different semantics/results in general

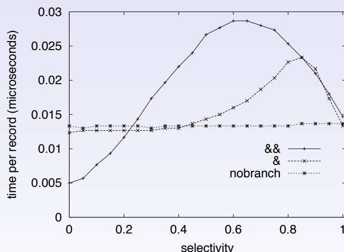


Figure 1: Three implementations: Pentium.

Cache-conscious algorithms - Selections

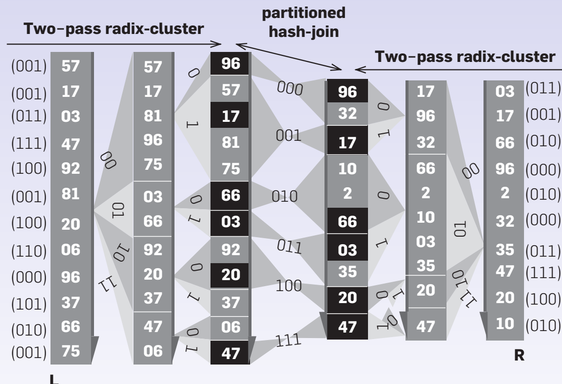
- Conjunctive selection conditions; Ross; PODS 2001
- Mixing the plans superior in many cases

```
/* A Mixed Algorithm (loop code omitted) */
  if((f1(r1[i]) & f2(r2[i])) && f3(r3[i]))
    { answer[j] = i;
      j += (f4(r4[i]) & ... & fk(rk[i]));
    }
```

Cache-conscious algorithms - Hash Joins

- Discussion from: "Breaking the Memory Wall in MonetDB"; CACM 2008
- First idea: Make each partition fit into a cache line, say a total of H clusters
- Problem: The partitioning itself creates a huge random access pattern
- Radix cluster:
  - A multi-pass algorithm to do the partitioning into H clusters

Cache-conscious algorithms - Hash Joins



Black tuples hit (lowest 3-bits of values in parenthesis)

Cache-conscious algorithms - Prefetching in Hash Joins

- Improving Hash Join Performance through Prefetching; TODS 2007
- Standard hash joins spend 80% time stalled on CPU cache misses
- Try to use "pre-fetching" to hide the cache misses
  - Most standard algorithms rely on identifying predictable patterns
  - Hash joins don’t generate predictable patterns, but we know the pattern
  - So optimizing at the algorithm level helps

Outline

PAX

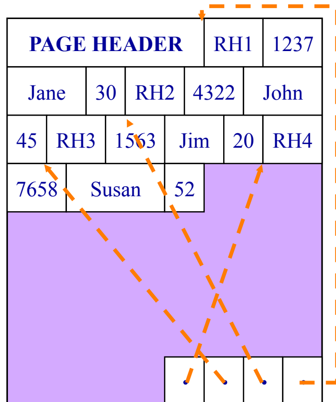
- Focus on cache misses and storage layout within a page

Current Scheme: Slotted Pages

Formal name: NSM (N-ary Storage Model)

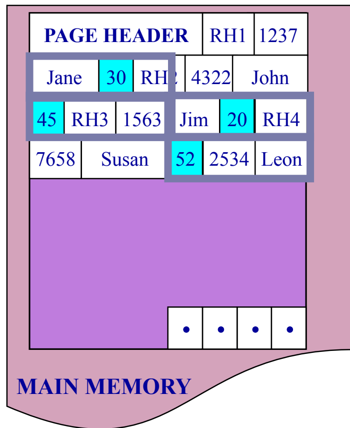
R

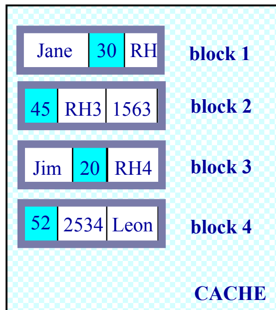
RID	SSN	Name	Age
1	1237	Jane	30
2	4322	John	45
3	1563	Jim	20
4	7658	Susan	52
5	2534	Leon	43
6	8791	Dan	37



- Records are stored sequentially
- Offsets to start of each record at end of page

Predicate Evaluation using NSM

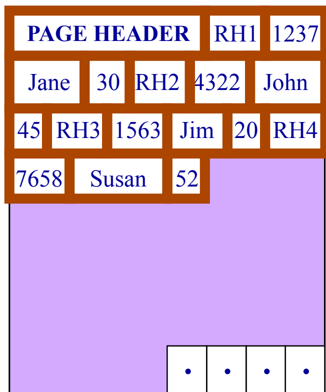


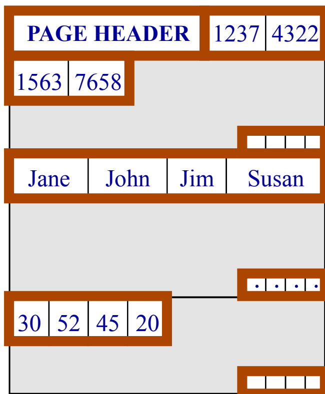



```
select name
from R
where age > 50
```

NSM pushes non-referenced data to the cache

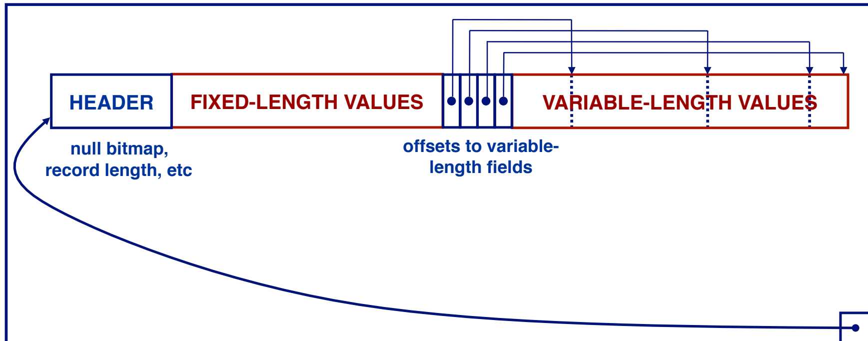
Partition Attributes Across (PAX)





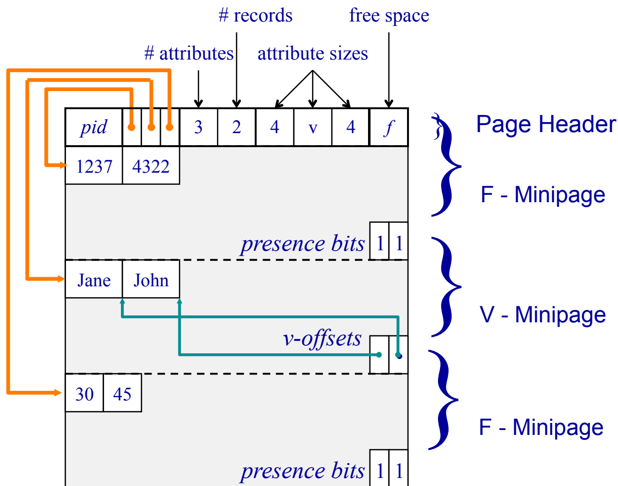
Partition data *within* the page for spatial locality

A Real NSM Record



NSM: All fields of record stored together + slots

PAX: Detailed Design



PAX: Group fields + amortizes record headers

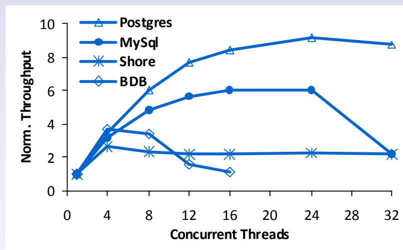
Outline

Multi-core

- Trend towards adding many cores on a single chip
  - Not really possible to increase the performace of a single CPU much more
- Typical today: many CPUs (multi-socket), each many-core
  - Often end up with a NUMA architecture
- Cores share L2, L3 caches
  - Can lead to cache pollution, contention
- Programming can be headache
- New bottlenecks not seen in previous CPUs with few cores
- Transactions are more problematic than read-only queries
  - Database operations are highly parallelizable so having enough work to do not a problem
  - But still need to be careful about NUMA architecture

Multi-core

- Discussion from: "Shore-MT: A scalable storage manager for the multicore era"; Johnson et al.; 2009
- Scalability experiments on 4 systems
  - No contention (each client creates a private table and inserts into it)



- Principles for scalable storage managers
  - Efficient synchronization primitives needed
  - Shorten or remove critical sections
  - Eliminate hot-spots, even read-only

Outline

GPUs

- Discussion from notes by Jens Teubner (ETH)
- GPUs have increasingly become general-purpose
- Programmability has increased dramatically
  - Originally: hard-coded fix-function pipeline
  - Today: C-like languages (e.g., CUDA, OpenCL)
- An early paper (SIGMOD 2004; Govindaraju et al.)
  - Use "stencils" (ability to render only parts of a screen) to evaluate Boolean predicates

GPUs

- Discussion from notes by Jens Teubner (ETH)
- Issues:
  - No direct access to GPU buffers from CPU
  - Data movement from host to GPU expensive
  - Limited memory on GPU
  - Programming often convoluted
  - Limited support for data types
- Many of these are addressed by now

GPUs

- Discussion from notes by Jens Teubner (ETH)
- GPUs provide data parallelism
  - Lightweight threads
  - 10,000 of threads of 100s of cores
  - All threads run the same code (SIMD-like)
- How to use GPUs?
  - Must combine CPUs and GPUs
  - CPUs copy data into GPU buffers
  - Invoke compute functions on GPUs

Sorting with GPUs

- Current GPUs have 10x higher main memory bandwidth and high data parallelism
- Sorting
  - Key problem, and served as a benchmark for many years
  - Originally Sort Benchmark, then MinuteSort, PennySort, JouleSort (how much per Joule of energy used)
- External sorting
  - Distributed-based: disjointly partition input file by sort attribute, sort each partition separately
  - Merge-based: create sorted runs from contiguous chunks, do a merge
  - In either case, need about $\sqrt{}$ memory (in blocks) to sort a large file in two passes
    - Can sort a terabyte file in few GBs of RAM
- Issues with CPU
  - Cache misses, compute intensive

GPUs

- Designed to execute geometric transformations on rectangular pixel array
- Capabilities
  - Data parallelism
    - 96 comparisons per clock cycles (NVIDIA 7800 GTX)
  - Instruction parallelism
    - 7800 GTX has 313 GFLOPS
  - Dedicated memory interface
    - Much higher peak bandwidths
  - Low memory latencies

GPUTeraSort: Flow Diagram

- Use both CPUs and GPUs for different things

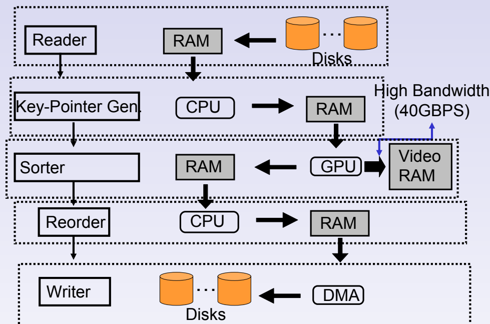


Figure 3: Flow Diagram of Phase 1 of GPUTeraSort Architecture using GPUs and CPUs.

GPUTeraSort

- Uses Bitonic sort
  - Highly parallel sorting algorithm
  - Standard algorithms like quicksort not a good fit because of dependencies
    - With bitonic sort, the same operations are executed independent of the data

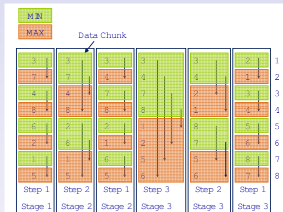


Figure 4: This figure illustrates a bitonic sorting network on 8 data values. The sorting algorithm proceeds in 3 stages. The output of each stage is the input to the next stage. In each stage, the array is conceptually divided into sorted data chunks or regions highlighted in green and red. Elements of adjacent chunks are merged as indicated by arrows. The minimum element is moved to the green region and the maximum is stored in the red colored regions producing larger sorted chunk.

- Shows huge benefits over CPU sorting