Motivation

The techniques we’ve seen so far all built on the same assumptions:

- Query processing cost is dominated by disk I/O.
- Main memory is random-access memory.
- Access to main memory has negligible cost.

Are these assumptions justified at all?
Hardware Trends

Source: Hennessy & Patterson, Computer Architecture, 4th Ed.

 normalized performance

 year

 Processor

 DRAM Memory

 © Jens Teubner · Data Processing on Modern Hardware · Fall 2011
Hardware Trends

There is an increasing gap between CPU and memory speeds.

- Also called the memory wall.
- CPUs spend much of their time waiting for memory.
Memory ≠ Memory

**Dynamic RAM (DRAM)**
- State kept in **capacitor**
- Leakage → **refreshing** needed

**Static RAM (SRAM)**
- **Bistable** latch (0 or 1)
- Cell state stable → no refreshing needed
DRAM Characteristics

Dynamic RAM is comparably slow.

- Memory needs to be refreshed periodically ($\approx$ every 64 ms).
- (Dis-)charging a capacitor takes time.

- DRAM cells must be addressed and capacitor outputs amplified.

Overall we’re talking about $\approx 200$ CPU cycles per access.
Under certain circumstances, DRAM can be reasonably fast.

- DRAM cells are physically organized as a 2-d array.
- The discharge/amplify process is done for an entire row.
- Once this is done, more than one word can be read out.

In addition,
- Several DRAM cells can be used in parallel.
  ⟷ Read out even more words in parallel.

We can exploit that by using sequential access patterns.
SRAM, by contrast, can be very **fast**.

- Transistors actively drive output lines, access almost **instantaneous**.

**But:**

- SRAMs are significantly more expensive (chip space $\equiv$ money)

**Therefore:**

- Organize memory as a **hierarchy**.
- Small, fast memories used as **caches** for slower memory.
### Memory Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Technology</th>
<th>Capacity</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>SRAM</td>
<td>bytes</td>
<td>&lt; 1 ns</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>SRAM</td>
<td>kilobytes</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>SRAM</td>
<td>megabytes</td>
<td>&lt; 10 ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>DRAM</td>
<td>gigabytes</td>
<td>70–100 ns</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Some systems also use a 3rd level cache.
- cf. Architecture & Implementation course
  - Caches resemble the buffer manager but are controlled by hardware
Caches take advantage of the principle of locality.

- 90% execution time spent in 10% of the code.
- The hot set of data often fits into caches.

Spatial Locality:
- Code often contains loops.
- Related data is often spatially close.

Temporal Locality:
- Code may call a function repeatedly, even if it is not spatially close.
- Programs tend to re-use data frequently.
To guarantee speed, the **overhead** of caching must be kept reasonable.

- Organize cache in **cache lines**.
- Only load/evict **full cache lines**.
- Typical **cache line size**: 64 bytes.

The organization in cache lines is consistent with the principle of (spatial) locality.

- Block-wise transfers are well-supported by DRAM chips.
On every memory access, the CPU checks if the respective cache line is already cached.

Cache Hit:
- Read data directly from the cache.
- No need to access lower-level memory.

Cache Miss:
- Read full cache line from lower-level memory.
- Evict some cached block and replace it by the newly read cache line.
- CPU stalls until data becomes available.\(^2\)

\(^2\)Modern CPUs support out-of-order execution and several in-flight cache misses.
In a fully associative cache, a block can be loaded into any cache line.

- Offers freedom to block replacement strategy.
- Does not scale to large caches
  - 4 MB cache, line size: 64 B: 65,536 cache lines.
- Used, e.g., for small TLB caches.
In a **direct-mapped** cache, a block has only one place it can appear in the cache.

- **Much** simpler to implement.
- Easier to make **fast**.
- Increases the chance of **conflicts**.
A compromise are set-associative caches.

- Group cache lines into sets.
- Each memory block maps to one set.
- Block can be placed anywhere within a set.
- Most processor caches today are set-associative.
Effect of Cache Parameters

Source: Ulrich Drepper. What Every Programmer Should Know About Memory

512 kB 1 MB 2 MB 4 MB 8 MB 16 MB

cache size

cache misses (millions)

direct-mapped
2-way associative
4-way associative
8-way associative

direct-mapped
2-way associative
4-way associative
8-way associative

512 kB 1 MB 2 MB 4 MB 8 MB 16 MB

cache size

Source: Ulrich Drepper. What Every Programmer Should Know About Memory
A **tag** associated with each cache line identifies the memory block currently held in this cache line.

The **tag** can be derived from the **memory address**.

```
status  tag  data
```

```
byte address

---
tag
| set index|
---|---|
| offset|
---
block address
```
Example: Intel Q6700 (Core 2 Quad)

- Total cache size: **4 MB** (per 2 cores).
- Cache line size: **64 bytes**.
  - → 6-bit offset \((2^6 = 64)\)
  - → There are 65,536 cache lines in total \((4 \text{ MB} \div 64 \text{ bytes})\).
- Associativity: **16-way set-associative**.
  - → There are 4,096 sets \((65,536 \div 16 = 4,096)\).
  - → 12-bit set index \((2^{12} = 4,096)\).
- Maximum physical address space: **64 GB**.
  - → 36 address bits are enough \((2^{36} \text{ bytes} = 64 \text{ GB})\)
  - → 18-bit tags \((36 - 12 - 6 = 18)\).

![Cache Address Layout Diagram]

© Jens Teubner · Data Processing on Modern Hardware · Fall 2011 32
Block Replacement

When bringing in new cache lines, an existing entry has to be evicted. Different strategies are conceivable (and meaningful):

Least Recently Used (LRU)
- Evict cache line whose last access is longest ago.
  \[ \rightarrow \text{Least likely to be needed any time soon.} \]

First In First Out (FIFO)
- Behaves often similar like LRU.
- But easier to implement.

Random
- Pick a random cache line to evict.
- Very simple to implement in hardware.

Replacement has to be decided \textbf{in hardware} and \textbf{fast}. 

What Happens on a Write?

To implement memory **writes**, CPU makers have two options:

**Write Through**
- Data is directly written to lower-level memory (and to the cache).
  - Writes will **stall the CPU**.\(^3\)
  - Greatly simplifies **data coherency**.

**Write Back**
- Data is only written into the cache.
- A **dirty** flag marks modified cache lines (Remember the status field.)
  - May reduce traffic to lower-level memory.
  - Need to write on eviction of dirty cache lines.

Modern processors usually implement **write back**.

---

\(^3\)Write buffers can be used to overcome this problem.
Putting it all Together

To compensate for slow memory, systems use caches.

- DRAM provides high capacity, but long latency.
- SRAM has better latency, but low capacity.
- Typically multiple levels of caching (memory hierarchy).
- Caches are organized into cache lines.
- **Set associativity**: A memory block can only go into a small number of cache lines (most caches are set-associative).

Systems will benefit from locality.

- Affects data and code.
Example: AMD Opteron

Example: AMD Opteron, 2.8 GHz, PC3200 DDR SDRAM

- **L1 cache**: separate data and instruction caches,
  each 64 kB, 64 B cache lines, 2-way set-associative

- **L2 cache**: shared cache,
  1 MB, 64 B cache lines, 16-way set-associative, pseudo-LRU policy

- **L1 hit latency**: 2 cycles

- **L2 hit latency**: 7 cycles (for first word)

- **L2 miss latency**: 160–180 cycles
  (20 CPU cycles + 140 cy DRAM latency (50 ns) + 20 cy on mem. bus)

- **L2 cache**: write-back

- 40-bit virtual addresses

Performance (SPECint 2000)

misses per 1000 instructions

- L1 Instruction Cache
- L2 Cache (shared)

benchmark program

gzip, vpr, gcc, mcf, crafty, parser, eon, perl, gap, vortex, bzip2, twolf, avg, TPC-C
Why do database systems show such poor cache behavior?

- Poor code locality:
  - Polymorphic functions (E.g., resolve attribute types for each processed tuple individually.)
- Volcano iterator model (pipelining): Each tuple is passed through a query plan composed of many operators.

- Poor data locality:
  - Database systems are designed to navigate through large data volumes quickly.
  - Navigating an index tree, e.g., by design means to "randomly" visit any of the (many) child nodes.
How can we improve data cache usage?

Consider, e.g., a selection query:

```
SELECT COUNT(*)
FROM lineitem
WHERE l_shipdate = "2009-09-26"
```

- This query typically involves a **full table scan**.
Tuples are represented as records stored sequentially on a database page.

- With every access to a `l_shipdate` field, we load a large amount of irrelevant information into the cache.
- Accesses to slot directories and variable-sized tuples incur additional trouble.
Motivation

Let’s have a look at a real, large-scale database:

- **Amadeus IT Group** is a major provider for travel-related IT.
- Core database: “Global Distribution System” (GDS):
  - dozens of millions of flight bookings
  - few kilobytes per booking
  - several hundred gigabytes of data

These numbers may sound impressive, **but**:

- The **hot set** of this database is significantly slower.
  - Flights with near departure times are most interesting.
- My laptop already has four gigabytes of RAM.

It is perfectly realistic to have the hot set in **main memory**.
Row-Wise Storage

Remember the row-wise data layout we discussed in Chapter I:

- Records in Amadeus’ ITINERARY table are ≈ 350 bytes, spanning over 47 attributes (i.e., 10–30 records per page).
Row-Wise Storage

To answer a query like

```
SELECT * FROM ITINERARY
WHERE FLIGHTNO = 'LX7' AND CLASS = 'M'
```

the system has to **scan** the entire ITINERARY table.¹⁸

- The table probably won’t fit into main memory as a whole.
- Though we always have to fetch full tables from disk, we will only inspect $\approx 20$–$60$ data items per page (to decide the predicate).

---

¹⁸ assuming there is no index support
Column-Wise Storage

Compare this to a column-wise storage:

We now have to evaluate the query in two steps:

1. **Scan** the pages that contain the FLIGHTNO and CLASS attributes.
2. For each matching tuple, **fetch** the 45 missing attributes from the remaining data pages.
Column-Wise Storage

- We read only a subset of the table, which may now fit into memory.
- We actually use hundreds or thousands of data items per page.
- **But:** We have to re-construct each tuple from 45 different pages.

Column-wise storage particularly pays off if

- tables are **wide** (*i.e.*, contain many columns),
- there is **no index support** (in high-dimensional spaces, *e.g.*, indexes become ineffective / Chapter III), and
- queries have a **high selectivity**.

**OLAP workloads** are the prototypical use case.
Example: MonetDB

The open-source database MonetDB\textsuperscript{19} pushes the idea of vertical decomposition to its extreme:

- All tables ("binary association tables, BATs") have 2 columns.

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>John</td>
<td>M</td>
</tr>
<tr>
<td>1723</td>
<td>Marc</td>
<td>M</td>
</tr>
<tr>
<td>6381</td>
<td>Betty</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4711</td>
</tr>
<tr>
<td>1</td>
<td>1723</td>
</tr>
<tr>
<td>2</td>
<td>6381</td>
</tr>
</tbody>
</table>

- Columns that carry consecutive numbers (such as OID above) can be represented as virtual columns.
  - They are only stored implicitly (tuple order).
  - Reduces space consumption and allows positional lookups.

\textsuperscript{19}http://www.monetdb.org/
Reduced Memory Footprint

- With help of column-wise storage, the **hot set** of the database may better fit into main memory.
- In addition, it increases the effectiveness of **compression**.
  - All values within a page belong to the same **domain**.
  - There’s a high chance of **redundancy** in such pages.
- So, with “all” data in main memory, are we done already?
Main Memory Access Cost

```cpp
int data[rows * columns];
for (int c = 0; c < columns; c++)
    for (int r = 0; r < rows; r++)
        process(data[r * columns + c]);
```

![Graph showing execution time vs. number of rows (rows × columns constant)]
Main Memory Access Cost

```c
int data[arr_size];
for (int i = arr_size - 1; i >= 0; i -= stride)
    process (data[i]);
```

- Memory access incurs a significant **latency** (209 CPU cycles here).
- (Multiple levels of) **caches** try to hide this latency.
- Latency is **increasing** over time.
Memory Access Cost

- Various **caches** lead to the situation that RAM is **not** random-access in today’s systems.
  - multi-level **data caches**
    (Intel x86: two levels\(^{20}\), AMD: three levels),
  - **instruction caches**,  
  - **translation lookaside buffers (TLBs)**
    (to speed-up virtual address translation).
- Novel database systems (sometimes called “main-memory databases”) include algorithms that are optimized for in-memory processing.
  - To keep matters simple, they assume that all data always resides in main memory.

\(^{20}\)The new i7 processor line has an L3 cache, too.
Optimizing for Cache Efficiency

To access main memory, CPU caches, in a sense, play the role that the buffer manager played to access the disk.

- Use the same “tricks” to make good use of the caches.
- Data processing in blocks
  - Choose block size to match the cache size now.
- Sequential access
  - Explicit hardware support for sequential scans.
- Use prefetching if possible.
  - E.g., x86 prefetchnta assembly instruction.
- What page size was in the buffer manager, is the cache line size in the CPU cache (e.g., 64 bytes).
In-Memory Hash Join

Straightforward clustering may cause problems:

- If $H$ exceeds the number of cache lines, cache thrashing occurs.
- If $H$ exceeds the number of TLB entries, clustering will thrash the TLB.

How could we avoid these problems?
Radix Clustering

$h_1$ and $h_2$ are the same hash function, but they look at different bits in the generated hash.
Radix Clustering

- SGI Origin 2000, 250 MHz, 32 kB L1 cache, 4 MB L2 cache.
Optimizing Instruction Cache Usage

Consider a query processor that uses tuple-wise pipelining:

- Each tuple is passed through the pipeline, before we process the next one.
- For eight tuples we obtain an execution trace
  \[ ABCABCABCABCABCABCABCABCABC \]
  where \( A, B, \) and \( C \) correspond to the code that implements the three operators.
- Depending on the size of the code that implements \( A, B, \) and \( C \), this can mean instruction cache thrashing.
Optimizing Instruction Cache Usage

- We can improve the effect of instruction caching if we do pipelining in larger chunks.
- *E.g.,* four tuples at a time:

  AAAABBBBCCCCAAAAABBBBCCCC

- Three out of four executions of every operator will now find their instructions cached.$^{21}$
- MonetDB again pushes this idea to the extreme. **Full tables** are processed at once ("full materialization").

What do you think about this approach?

---

$^{21}$This assumes that $A$, $B$, and $C$ fit into the instruction cache individually. A variation is to group operators, such that the code for each group fits into cache.
Multiple CPU Cores

- Current trend in hardware technology is to no longer increase clock speed, but rather increase parallelism.
- Multiple CPU cores are packaged onto a single die.
- Such cores often share a common cache.
  - If such cores work on fully independent tasks, they will often compete for the shared cache.
- Can we make them work together instead?
Bi-Threaded Operators

Idea: Pair each database thread with a helper thread.

- All operator execution remains in the main thread.
- The helper thread works ahead of the main thread and preloads the cache with data that will soon be needed by the main thread.
- While the helper thread experiences all the memory stalls, the main thread can continue doing useful work.

Work-Ahead Set

Main and helper thread communicate via a work-ahead set.

- **Main thread posts** soon-to-be-needed memory references $p_i$ into a work-ahead set.
- **Helper thread reads** memory references $p_i$ from the work-ahead set, accesses $p_i$, and thus populates the cache.

```
main thread  →  post(\(p\))  →  helper thread
\[ \text{w.-a. set} \]
\[ p_1 \]
\[ p_2 \]
\[ p_3 \]
\[ \vdots \]

shared cache
```
Work-Ahead Set

- Note that the correct operation of the main thread does not depend on the helper thread.

Why not use CPU-provided prefetch instructions instead?

- Prefetch instructions (e.g., prefetchnta) are only hints to the CPU, which are not binding.
- The CPU will drop prefetch requests, e.g., if prefetching would cause a TLB miss.

- Bi-threaded operators need to be implemented with care.
  - Concurrent access to the work-ahead set may cause communication between CPU cores to ensure cache coherence.
Heterogeneous Multi-Core Systems

- In addition to an increased number of CPU cores, there is also a trend toward an increased **diversification** of cores, *e.g.*, 
  - **graphics processors** (GPUs),
  - **network processors**.
  - The **Cell Broadband Engine** comes with one general-purpose core and eight “synergetic processing units (SPEs)”, optimized for vector-oriented processing.

- Some of their functionality is well-suited for expensive database tasks.
  - **Sorting**, *e.g.*, can be mapped to GPU primitives.  
  - Network processors provide excellent **multi-threading** support.