

## Part VII

# Databases on Modern Hardware

# 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?

# 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

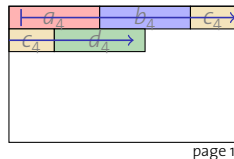
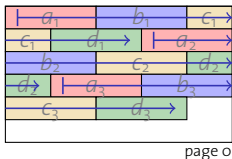
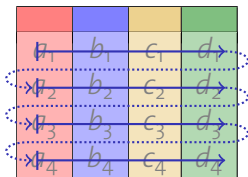
These numbers may sound impressive, **but**:

- ▶ 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 1:



- ▶ Records in Amadeus' ITINERARY table are  $\approx 350$  bytes, spanning over 47 attributes (*i.e.*, 10–30 records per page).

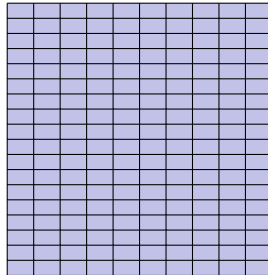
# Row-Wise Storage

To answer a query like

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

the system has to **scan** the entire ITINERARY table.<sup>18</sup>

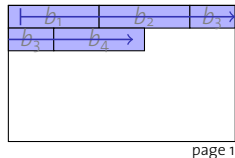
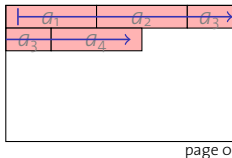
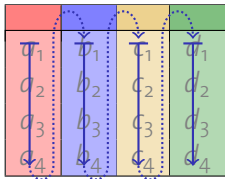
- ▶ The table probably won't fit into main memory as a whole.
- ▶ Though we always have to fetch full tables from disk, we will only inspect  $\approx 20$ – $60$  data items per page (to decide the predicate).



<sup>18</sup>assuming there is no index support

# Column-Wise Storage

Compare this to a column-wise storage:



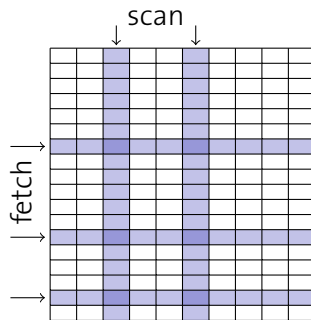
...

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**<sup>19</sup> pushes the idea of **vertical decomposition** to its extreme:

- ▶ All tables (“binary association tables, BATs”) have **2 columns**.

ID	NAME	SEX
4711	John	M
1723	Marc	M
6381	Betty	F

~>

OID	ID
0	4711
1	1723
2	6381

OID	NAME
0	John
1	Marc
2	Betty

OID	SEX
0	M
1	M
2	F

- ▶ 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**.

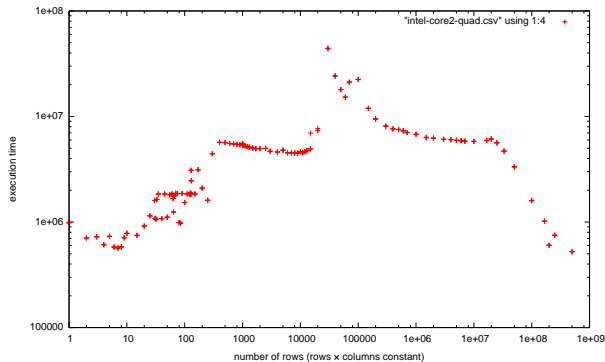
<sup>19</sup><http://www.monetdb.org/>

# 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

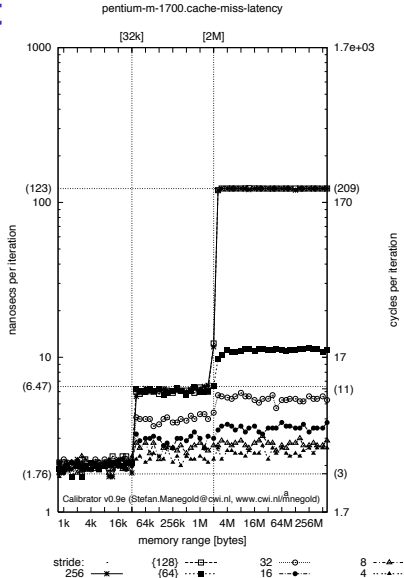
```
int data[rows * columns];
for (int c = 0; c < columns; c++)
    for (int r = 0; r < rows; r++)
        process (data[r * columns + c]);
```



# Main Memory Access Cost

```
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<sup>20</sup>, 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.

---

<sup>20</sup>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).

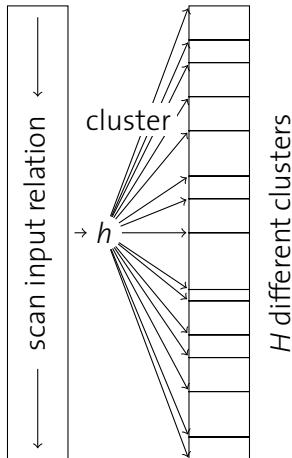
# In-Memory Hash Join

Straightforward clustering may cause problems:

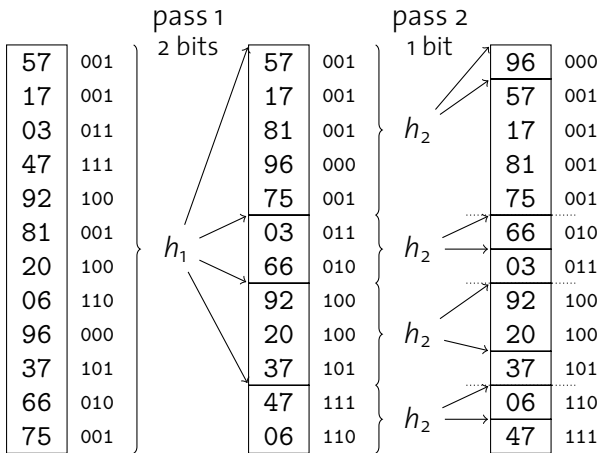
- ▶ If  $H$  exceeds the number of **cache lines**, **cache thrashing** occurs.
- ▶ If  $H$  exceeds the number of **TLB entries**, clustering will thrash the TLB.



How could we avoid these problems?

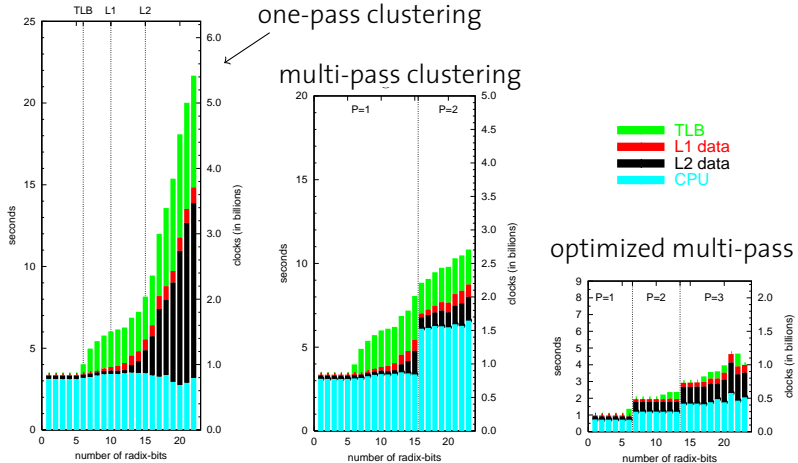


# Radix Clustering



- ▶  $h_1$  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.  
 ↗ S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. IEEE TKDE, vol. 14(4), Jul/Aug 2002.

# 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

*ABCABCABCABCABCABCABC* ,

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.

⋮  
C  
|  
B  
|  
A  
⋮

# 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:

AAAABBBBCCCCAAAABBBBCCCC .

- ▶ Three out of four executions of every operator will now find their instructions cached.<sup>21</sup>
- ▶ MonetDB again pushes this idea to the extreme. **Full tables** are processed at once (“full materialization”).

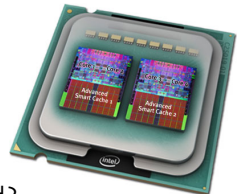


**What do you think about this approach?**

<sup>21</sup>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**.

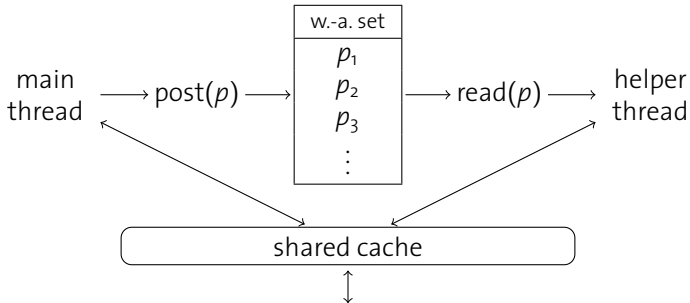
- ▶ All operator execution remains in the main thread.
- ▶ The helper thread **works ahead** of the main thread and **preloads** the cache with data that will soon be needed by the main thread.
- ▶ While the helper thread experiences all the memory stalls, the main thread can continue doing useful work.

↗ J. Zhou, J. Cieslewicz, K. A. Ross, M. Shah. Improving Database Performance on Simultaneous Multithreading Processors. VLDB 2005.

# 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.



# 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.  
↗ Govindaraju *et al.* GPUteraSort: High Performance Graphics Co-Processor Sorting for Large Database Management. SIGMOD 2006.
  - ▶ Network processors provide excellent **multi-threading** support.