...like Commanding an Anthill: A Case for Micro-Distributed (Data) Management Systems

Holger Pirk
MIT CSAIL, Cambridge, USA
holger@csail.mit.edu

ABSTRACT

Computer system architecture has changed: an assembly of autonomous components has replaced the omnipotent CPU and its legion of dumb devices. Database Management System (DBMS) architecture, however, does not yet reflect this change: it is still dominated by a centralized kernel that limits the autonomy of the devices and, thus, their ability to exploit their increased "smartness". Distributed data management research can serve as an inspiration for an architecture that addresses this problem. However, the respective algorithms were never designed with CPU efficiency in mind implementing principles like dynamic programming and recursion.

More than two decades ago, the transition to memory resident databases spawned a plethora of research on CPU-efficient query processors. We predict that hardware heterogeneity will trigger a similar line of research on CPU-efficient distributed algorithms and architectures. In this paper, we examine benefits and challenges that come with such a *micro-distributed* database management system. We also discuss a number of approaches that we consider steps towards a micro-distributed system.

1. INTRODUCTION

Modern computer systems are not the centralized machines they used to be: they are an assembly of autonomous components. While connected through relatively simple interfaces, these systems incorporate quite intricate control logic. Modern Solid State Disk (SSD) firmwares, e.g., have thousands of lines of code and implement features such as garbage collection, buffer management and request queuing [6]. This stands in stark contrast to the assumptions that classic data management system design was based on: a *Central* Processing Unit (CPU) with autocratic powers that micro-manages the program execution flow. In such an architecture, subsystems and devices get specific instructions (move arm, read sector, perform arithmetic operation) from the CPU and any delay in execution causes stall cycles in the CPU which hurts performance. Increased execution autonomy could, however, be used by the devices to increase performance, energy efficiency or even lifespan. Aforementioned SSD firmwares, e.g., combine multiple Out-of-Order (OOO) writes which reduces overhead when writing to memory cells and improves wear-leveling. Similarly, massively parallel co-processors merge multiple sequential programs into a single (massively) parallel program.

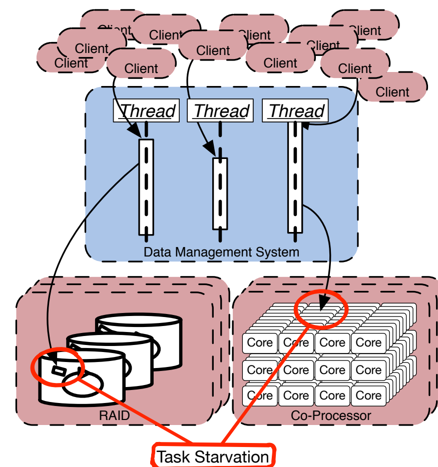


Figure 1: The DBMS: a Concurrency Bottleneck

However, all of these benefits hinge on a single point: the software has to provide the necessary degree of autonomy to each of the subsystems. While device autonomy has a number of dimensions [11], let us focus on execution autonomy for now:

a *device with execution autonomy* has the freedom to execute operations in any way or order it sees fit.

To ensure utilization there have to be enough operations to choose from at any given time. Unfortunately, the required OOO degree is highly device specific: SSD command queue length is currently limited to 32 by the SATA specification but in-

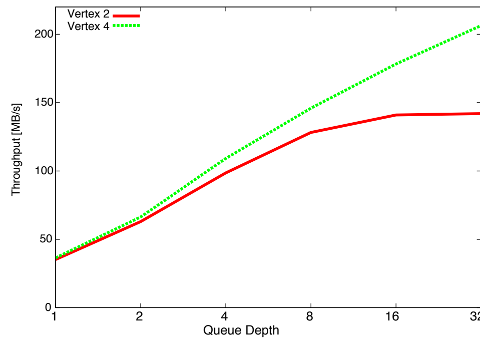


Figure 2: Impact of out-of-order degree on SSD performance (taken from [15])

creasing to many thousands in next-generation devices [4]; co-processor cards already support hundreds to thousands of hardware threads. To encompass all of these as well as future devices, it is common to go for maximum, i.e., massive (data) parallelism (e.g., [10]). Unfortunately this approach is not applicable to task parallelism: even the degree of parallelism of current parallel devices cannot be sustained by the centralized components of classic data management systems (see Figure 1) leading to *Task Starvation*. *Task Starvation*, i.e., a lack of independent work items, leads to insufficient optimization opportunities and, thus, suboptimal performance of hardware devices (we dicuss this point in more detail in Section 2.1).

DBMSs are not the only systems suffering from this problem. In fact, they have inherited it from the underlying component in the application stack: the operating system. The root of the problem lies in preemptive multitasking and the associated context thrashing: to micro-manage an autonomous device, a thread has to be frequently awoken, its context be loaded, the device status checked, the context persisted and the tread retired. This causes substantial overhead creating a scalability bottleneck.

This problem, however, is not entirely new to data management systems: distributed Database Management Systems (DBMSs) by definition deal with autonomous, remote subsystems. When working with such autonomous subsystems, the system cannot make assumptions about the order, timeliness or quality of the result [11]. We argue that, to withstand the challenges of the age of massively parallel hardware, centralized databases systems have to become equally robust against autonomous behavior as their distributed cousins: become *micro-distributed*.

Unfortunately, transplanting ideas from the domain of distributed databases into centralized systems is not entirely trivial. We will dedicate the next two sections to a discussion of benefits (Section 2) and challenges (Section 3) of such an approach. We will also provide a brief survey of the approaches that may be steps towards a micro-distributed data management system in Section 4. We will conclude with a brief outlook in Section 6

2. BENEFITS

The primary benefit of a micro-distributed DBMS is, naturally, the increase in performance and scalability that comes with increased execution autonomy. However, there are a number of secondary benefits that we want to cover in this section.

2.1 Performance

Naturally, the performance impact of micro-distribution depends on the degree to which the targeted device can exploit the OOO-degree. It is, therefore, hard to make substantiated predictions about the performance impact. However, there are studies that can help to gain an impression of the performance impact that high OOO-degrees can have on device performance.

Figure 2 shows the result of a simple micro-experiment: random read requests to a single SSD (comparing devices with different internal OOO-degrees). The experiment shows a steady logarithmic increase in transfer rate when scaling the degree of parallelism. While this experiment was conducted using a hand-coded driver on an FPGA, it demonstrates the potential of execution autonomy in the hardware. In current-generation SATA-SSDs the OOO-degree is limited to 32. Next generation non-volatile-ram devices have, by specification, an OOO-degree of 65,536. This illustrates the need for more OOO-parallelism in the DBMS.

2.2 Secondary Benefits

Besides the performance benefits through device autonomy, there are a number of secondary benefits inherited from classic distributed systems.

A Unified Architecture

Most industry grade DBMSs support distributed as well as centralized databases. However distributed data management is usually implemented in a separate module that is only activated if needed. The core, i.e., centralized, query processor usually follows the "autocratic" model we described earlier while the distributed module is implemented under the assumption that remote systems have substantial autonomy.

What we propose essentially merges the two subsystems. While there were good reasons to separate them (see Section 3), unifying them promises a significant reduction in system complexity.

Separation of Concerns

An increase in autonomy not only enables devices to execute multiple requests OOO, it also allows them to evaluate complex operations completely autonomous. This evidently holds for Graphics Processing Unitss (GPUs), which became programmable almost a decade ago. However, recent work turned even storage devices such as SSDs into *Smart Devices*, i.e., devices with builtin processing functionality. They can be used to efficiently perform operations such as prefiltering [1], aggregation [16] and even document ranking [12] without CPU-involvement. However, such functionality further aggravates the latency of involved storage devices.

Device Hot-Swapping & Virtualization

One of the appealing traits of distributed databases is the capability to add and remove subsystems at runtime. A micro-distributed system extends this capability to centralized DBMSs. This capability allows, e.g., the virtualization of co-processors - a service that cloud services like Amazon's EC2 do not yet provide. In a multi-tenant setup, e.g., the system could swap in an accelerator card to deal with load spikes at much lower (transfer) costs than scaling out to an additional node.

3. CHALLENGES

When considering the benefits, it seems surprising why centralized and distributed database kernels ever became separated. The reason lies in the effort of managing of autonomous subsystems. We dedicate this section to a discussion of the challenges that come with a (re)unification of centralized and distributed kernels.

3.1 Efficiency

As indicated previously, distributed data management research can provide a good starting point for work towards micro-distributed systems. Naturally, the amount of research focusing distributed data management is staggering ([11] provides the background for much of our discussion). Distributed data management addresses a number of specific challenges such as distributed concurrency control, data and query placement, transfer-conscious processing and schema diversity. The CPU-efficiency of the underlying algorithms, however, is not usually considered an important factor. This has two main reasons:

- the number of concurrent queries usually run in an order that does not pose much of a scalability problem (hundreds rather than tens of thousands) and
- computational resources are usually abundant because the query costs are dominated by data transfer over the network.

These do not hold for micro-distributed DBMSs. In fact, we believe scalable CPU-efficiency to be the main challenge for micro-distributed systems.

When in-memory data management solutions became feasible due to falling prices, they faced a similar situation: the obvious first step was to simply run existing software on memory-resident data - Data structures (tree-indices, slotted pages, ...) as well as system architectures (buffer managers, volcano-style query processors, ...) were left untouched. However, it quickly became apparent that the CPU overhead of these techniques became the dominating cost factor [7]. It took almost two decades of research to increase CPU efficiency to a point at which the memory is the dominating bottleneck again [3]. We believe similar challenges lie ahead in micro-distributed systems.

3.2 Finding Appropriate Work

While autonomous devices are to a large extend programmable, they still have limitations. GPUs, e.g., have thousands of processing cores but only few instruction schedulers. For that reason, the cores in one processing unit (between 32 and 128) have to share an instruction scheduler. This restricts the program to only a few (around 50) independent instruction streams. Similarly, smart SSDs can evaluate range predicates on integer values and calculate (simple) hashes but cannot provide functionality like floating point arithmetic or sorting. It will be a challenge for DBMS to break down complex queries into executable sub-tasks and, if need be, re-program the devices. Finding the right level of abstraction to "software-define" devices will be important.

3.3 Conveying High-Level Knowledge

With the limitations of the the devices also comes a lack of high-level understanding of the problem at hand. While the DBMS can keep track of high-level information such as operator dependencies or characteristics, it will be hard to convey these insights to the devices. On the data management side, this has the potential to spawn research on the creative use of existing functionality. On the hardware side, it can fuel work on novel interfaces and optimizations.

4. THE STATE OF THE ART

Before concluding, we want to briefly discuss promising steps towards a micro-distributed DBMS.

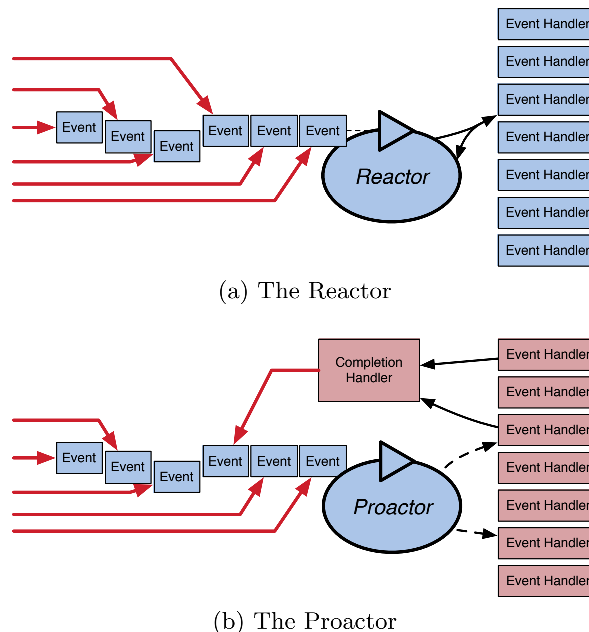
4.1 Data-Parallel Systems

Since the primary goal of micro-distribution is an increase in the OOO-degree in the underlying hardware, it is sensible to look at the most obvious source of parallelism: data. In fact, data parallelism is comparatively easy to manage since it is usually localized within a kernel operator. Data management on GPUs, e.g., has exploited data parallelism for almost a decade [9]. However, the data parallel approach falls short for operations that are not inherently data parallel such as index accesses or transaction processing. Consequently, data-parallel micro-distribution is currently limited to analytical workloads.

4.2 Multi-Kernel Approaches

We are by no means the first to recognize the convergence of modern computer architecture and distributed system design. Consequently, related work can be found in other fields of computer science research. In the field of operating systems, for example, the concept of the *Multi-Kernel OS* has recently been introduced [2]. The idea is to overcome multicore scalability issues of user applications by running an OS-Kernel instance per core and statically assigning execution threads to kernel instances. Any inter-thread communication is performed using asynchronous message passing. Existing message passing implementations, however, were designed for comparatively rare inter-machine or inter-process communication. The sheer amount of traffic that is generated by Multi-Kernel applications justifies a reimplementation of the message passing concept - aware of factors like Cache Line Sizes, Coherence Protocols and Non-Uniform Memory Access [2]. The existing work on Multi-Kernel OSes indicates comparable performance yet better scalability than existing operating systems. This is exactly the kind of benefits we strive for in the database domain.

A recent piece of work [14] has performed the first step towards such a micro-distributed data management system: a direct port of a classic distributed system. We believe this to be a step similar to the first direct ports of disk-resident data management kernels to memory-resident databases: a compelling case for the concept in a very restricted scenario (a replicated shared-nothing or perfectly partitionable database&workload). It seems unlikely that such a simple port will be competitive under less partitionable workloads. Just like a Multi-Kernel OS is fundamentally different from multiple Single-Kernel OS kernels, a micro-distributed DBMS is different from multiple federated database instances on a single node. We believe that, just like in operating systems, the key is lightweight message passing. Fortunately, there is a precedent for systems built around the idea of highly concurrent lightweight message passing as well: *Reactive Systems*.



(a) The Reactor

(b) The Proactor

Figure 3: The Design Patterns of Reactive Systems

4.3 Reactive Systems

Reactive systems implement the *Reactor Design Pattern* [5] (see Figure 3a). This Pattern was developed to address the high level of concurrency in the hardware components of communication systems which usually only support moderate parallelism in hardware. But also many highly concurrent networking software packages such as Twisted or Node.js implement the reactor pattern.

The principal idea is that requests are sequentialized by a single, lightweight *Reactor* thread and subsequently dispatched to *Event Handlers* which perform the actual work. Such reactive systems replace preemptive multithreading with a form of cooperative multithreading. This leads to much better scalability for task-parallel applications as long as all threads cooperate.

However, this points out at a crucial point: *Event Handlers* must only take a relatively short amount of time to process an *Event*. Since the *Reactor* is blocked while a *Handler* is active, the system be-

comes unresponsive if a *Handler* takes a long time to complete.

Therefore, many reactive systems actually implement a combination of the *Reactor Pattern* and the very closely related *Proactor Pattern* [13]. The main difference between the two is the concurrency model. In a *Proactor* implementation (see Figure 3b), the *Event Handlers* do not "borrow" the execution thread of the *Reactor* [13]. Instead, *Event Handlers* are invoked with a *Completion Handler* that is invoked by the *Event Handler* once it is done processing a particular *Event*. This makes the *Proactor* more suited for handling (relatively) long running operations because the *Reactor* thread is not blocked and can continue to dispatch *Events*. The *Proactor Pattern* effectively re-parallelizes the sequentialized events by dispatching them to concurrently running *Event Handlers*. This pattern efficiently multiplexes the concurrent events of the parallelization bottleneck (the task-sequential device) and reparallelizes them when appropriate (before processing them using the task-parallel device).

5. A REACTIVE DBMS

Earlier, we identified preemptive multithreading as a major scalability bottleneck when managing micro-distributed devices. Since the *Reactor Pattern* was designed for this very problem, it is sensible to consider a *Reactive Data Management System* as an appropriate response to this challenge. We, thus, want to use this section to outline the design of such a system.

The Reactor. Most reactive systems implement the *Reactor* in some kind of *Event Loop* that processes occurring *Events* in the order in which they are enqueued. Naturally, a reactive DBMS would also incorporate such an *Event Loop* (see Figure 4). However, since the *Event Loop* is merely responsible for dispatching work to the appropriate *Handlers*, this component is very lightweight. The actual component logic is encapsulated in the *Event Handlers*.

The Event Handlers. All of the *actual* functionality of a reactive DBMS is implemented in the *Event Handlers*. This includes classic components like Optimization, Recovery or Buffer Management. However, it also includes all data access operations which are usually part of high-level operators.

Reactive Query Processing.

A typical select & project query on a column oriented database, e.g., is likely to go through a high number of *Event Handlers*: imagine starting with

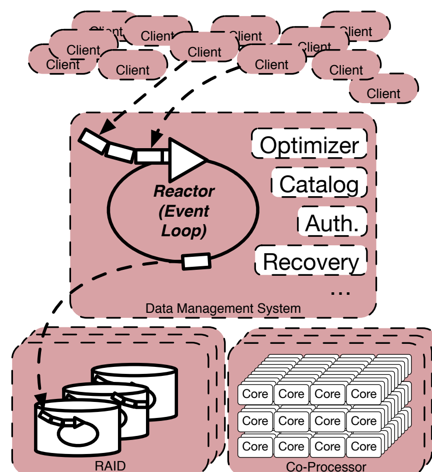


Figure 4: A Reactive Data Management System Architectures

a GPU-based handler performing massively parallel scan of the selection-column to retrieve the ID(s) of matching tuple(s); after that, every projected attribute could result in at least one call to a project-handler touching disk-resident columns. Even potential accesses to secondary index structures such as Primary- or Foreign-Key indices are processed using individual *Events* with every access to a node of tree-based index constituting an event. By taking the concept to these extremes, we can maximize the number of outstanding SSD requests and, thus, the resulting optimization opportunities.

This indicates one of the greatest challenges in the development of a *Reactive DBMS*: event processing overhead. While asynchronicity helps to increase the OOO-degree available to devices, the CPU overhead of the event processor can result in significant costs. A potential solution could be to have multiple *Event Loops* hardcoded for a specific class of operations such as tree lookups or lock managing. In doing so, we approach the design of an interesting artefact of related work: *StagedDB* and the underlying idea of *one operator - many queries* [8].

One Operator - Many Queries.

StagedDB is, to the best of our knowledge, the first system to part from the *one query - many operators* model, replacing it with a *one operator - many queries* approach. The idea is to have a single, static operator instance per "operator class" rather than an instance per query plan node. Such a design improves instruction cache locality and allows the system to exploit occuring work-sharing opportunities. However, each operator identifies sharing

opportunities itself and uses synchronous interfaces to the underlying hardware. This not only increases complexity and makes performance vulnerable to changes of hardware parameters, it also spends significant resources on identifying sharing opportunites which often counteracts the benefits.

In contrast to *StagedDB*, our reactive approach does not invest effort in identifying sharing opportunities. Instead, it generates many OOO-request and leaves the identification of sharing opportunities to the hardware itself. In doing so, a reactive DBMS not only removes the substantial overhead of work-sharing analysis from the CPU but pushes it to the component that is specifically optimized for such analysis: the hardware device controller. *StagedDB* may, however, serve as a blueprint to mitigate the CPU overhead of a reactive DBMS by clustering low-level operations into static groups.

6. CONCLUSION

We have argued that the increasing heterogeneity of computer systems makes them more akin to distributed systems than their centralized ancestors. We believe that, to efficiently manage such heterogeneous systems, centralized DBMSs have to adopt distributed data management techniques. However, the need for CPU-efficiency makes a direct port of these techniques unfeasible. Like memory-resident data management triggered an era of research on CPU-efficient query processing, we believe that heterogeneity will spawn an era of research on CPU-efficient distributed algorithms. We have provided indications that this era has already begun but also provided a number of challenges that still lie ahead.

7. REFERENCES

[1] Acharya, A., Uysal, M., and Saltz, J. Active disks: Programming model, algorithms and evaluation. In *ACM SIGOPS Operating Systems Review* (1998), vol. 32, ACM.

[2] Baumann, A., Barham, P., Dagand, P.-E., et al. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM.

[3] Boncz, P. A., Kersten, M. L., and Manegold, S. Breaking the memory wall in monetdb. *Communications of the ACM 51*, 12 (2008).

[4] Cobb, D., and Huffman, A. Nvm express and the pci express ssd revolution, 2012.

[5] Coplien, J., and Schmidt, D., Eds. *Pattern Languages of Program Design.* Addison-Wesley, 1995, ch. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching.

[6] Cornwell, M. Anatomy of a solid-state drive. *Commun. ACM 55*, 12 (2012).

[7] Garcia-Molina, H., and Salem, K. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on 4*, 6 (1992).

[8] Harizopoulos, S., Ailamaki, A., et al. Stageddb: Designing database servers for modern hardware. *IEEE Data Engineering Bulletin 28*, 2 (2005), 11–16.

[9] He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS) 34*, 4 (2009).

[10] Ladner, R. E., and Fischer, M. J. Parallel prefix computation. *Journal of the ACM (JACM) 27*, 4 (1980).

[11] Özsu, M. T., and Valduriez, P. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[12] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE.

[13] Pyarali, I., Harrison, T., Schmidt, D. C., and Jordan, T. D. Proactor-an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. *Proceedings of the 4th Annual Pattern Languages of Programming Conference* (1997).

[14] Salomie, T.-I., Subasu, I. E., Giceva, J., and Alonso, G. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the sixth conference on Computer systems* (2011), ACM.

[15] Sidler, D. Column storage for fpga-accelerated data analytics. Master's thesis, ETH Zürich, 2013.

[16] Woods, L., István, Z., and Alonso, G. Ibexan intelligent storage engine with support for advanced sql off-loading. *Proc. VLDB Endowment (PVLDB)* (2014).