A Short Study of Recent Smart Storage Solutions for OLAP: Lessons and Opportunities

Faeze Faghih
Systems Group
Technical University of Darmstadt
Darmstadt, Germany
faeze.faghih@tu-darmstadt.de

Zsolt István
Systems Group
Technical University of Darmstadt
Darmstadt, Germany
zsolt.istvan@tu-darmstadt.de

Florin Dinu
Huawei Munich Research Center
Munich, Germany
florin.dinu@huawei.com

ABSTRACT

As the usage of data-intensive applications is increasing, moving data between the storage and host is becoming an important bottleneck. Smart storage has emerged as a solution to mitigate the data-movement bottleneck, resulting in lower latency, better energy efficiency, and better overall throughput. The last decade has seen a proliferation of proposals for architectures for smart storage.

In this short study, we look at a representative sample of these works and distill common lessons and techniques from them: first, high selectivity in itself is not a guarantee for better performance, second, it is important to match the chosen in-storage data layout with that of the processing element and application and third, leveraging heterogeneous processing elements inside smart storage benefits efficiency and flexibility.

Our main goal in compiling and sharing the list of lessons is to avoid reinventing the wheel when designing future smart storage solutions. At the same time, we also identify aspects that will be necessary for building future smart storage that can cater to varied workloads but are not well covered in related work today.

1 INTRODUCTION

Processing large volumes of data is common in database applications, especially in those serving analytic queries looking for and identifying patterns, discovering hidden relationships in the data, or deriving aggregate values and statistics. Since large datasets often do not fit in main memory, the host must load data from storage to answer queries and this data movement is an important bottleneck in modern database systems. One of the solutions for this bottleneck is bringing the computation near or inside the storage. By pre-processing data close to the source and transferring only a fraction of the original dataset to the host, data movement can be reduced significantly. This leads to improved performance and often also increased energy efficiency. In addition, by reducing the load on the host CPU and memory, such offloading techniques free up host resources that can be used to run more parallel queries or additional compute-intensive tasks.

There is a rich body of related works for bringing computation near data – with a history reaching back to more than a quarter of a century [5, 18]. In this study, we do not consider a "historical" perspective and instead focus on the representative projects of the last decade. The reason for this is that the slowing down of Dennard Scaling [11] means that the relative bandwidth/compute capacities these works consider are comparable to each other, as opposed to work from the early 1990s, for instance, that operated at orders of magnitude lower bandwidth and CPU performance.

The surveyed related work proposes an exciting variety of deployment options for processing elements in the storage device: it is possible to use the embedded CPU of the disk for running offloaded computation [9, 16, 25], insert a specific CPU inside the disk for running the in-storage computation [13], using processing elements based on Field Programmable Gate Arrays (FPGAs) specifically designed for this purpose [21, 22, 27], or even a combination of an embedded CPU and a specialized processing element [8, 12, 26].

Despite the fact that smart storage is a mature field, after surveying the related work, we still found that lessons and opportunities are often implicit or formulated specific to a workload/query combination. In this work, we highlight lessons and opportunities for building future smart storage solutions, applicable to novel, not yet implemented, operators. SQL query acceleration has been studied in detail and it is unclear what further improvements can be achieved – with emerging machine learning workloads [14, 17] and other types of computationally intensive tasks [10, 23] however, new opportunities emerge. In this context, it is important to take stock of what the related work can teach us and avoid reinventing the wheel to ensure that the new solutions will be as useful and efficient as possible. Therefore, we identified three principles for designing the next smart storage device:

(1) **Balance the compute cost of offloading against the data selectivity**: It is well understood that offloading to smart storage is (most) useful for operators that can significantly reduce data movement but, as the related work demonstrates, the choice of offloaded operations has to also take into account their compute requirements. Since smart storage solutions have less compute capacity than host CPUs, offloading too compute-intensive tasks might result in an overall slowdown even if they filter out large portions of the data. Since most related work considers only operators running alone on the smart storage, further investigation is necessary for making robust run-time scheduling decisions.
(2) **Match the data layout with the expected layout of the application and processing element**: Specialized implementations of operators, especially those relying on FPGAs, will work well only if they are designed with the right data layout in mind. Related work shows that, in case the data is laid out,

Table 1: Overview of studied smart storage solutions from the last decade

Name (year)	Platform	Processing Element	Evaluated Applications	Evaluated Queries	Data	Bandwidth to/from storage	Processing element location
Wisconsin Smart SSD [9] (2013)	SAS SSD	Embed. CPU (shared with SSD controller)	SQL operators	2 custom + 1 TPC-H	custom+ TPC-H (110 GB)	550 MB/s	co-processor
Ibex [27] (2014)	Custom FPGA with SATA SSD	FPGA	SQL operators	6 custom + 1 TPC-H	TPC-H (10 GB)	300 MB/s, 125 MB/s	in datapath
Biscuit [12] (2016), YourSQL [15] (2016)	Commercial SSD	Embed. CPU (shared with SSD controller) + ASIC	SQL operators, pointer chasing	custom + all TPC-H queries	TPC-H (160 GB)	PCIe Gen3 x4 (3.2 GB/s)	in datapath (ASIC), co-processor (CPU)
Summarizer [16] (2017)	LS2085a intelligent SSD (running at 200MHz)	Embed. CPU (shared with SSD controller)	SQL operators, data integration	2 custom + 3 TPC-H	TPC-H (100 MB)	PCIe Gen3 x4 (3.9 GB/s)	co-processor
Insider [20] (2019)	Drive Prototype	FPGA	SQL operators, decompression, KNN	1 custom	custom (60 GB)	PCIe Gen3 x8,x16 (7.8,15.7 GB/s)	on the side
AQUOMAN [28] (2020)	Simulation and FPGA prototype	FPGA	SQL operators	all TPC-H queries	TPC-H (1 TB)	PCIe	on the side
FANS [19] (2021), NASCENT [21] (2021)	Samsung SmartSSD	FPGA	Bitonic Merge Sort	-	(Terasort [4], 2^{33}elements) ,(random, 2^{10}-2^{13} elements)	PCIe Gen3 x4	on the side
Newport [13] (2022)	NGD Systems Newport SSD	Dedicated Embed. CPU	SQL operators, distributed training-inference	YCSB	-	PCIe	on the side

e.g., in a columnar fashion in storage instead of row-oriented fashion (or vice versa), the data transformations can consume significant additional resources [24], reducing the overall performance gains significantly.

(3) **Embrace heterogeneity**: Most of the surveyed related works rely on either a low-power CPU or a fully specialized ASIC/FPGA for processing offload. Given the different tasks required for successful offloading (data parsing, number crunching, metadata processing, etc.) we believe that future solutions should have heterogeneous processing elements (e.g., as in Biscuit [12]). The main challenge of using such architectures, however, is increased scheduling and resource allocation complexity. Luckily, the database community is well prepared in overcoming such challenges!

2 SUMMARY OF SELECTED RELATED WORK

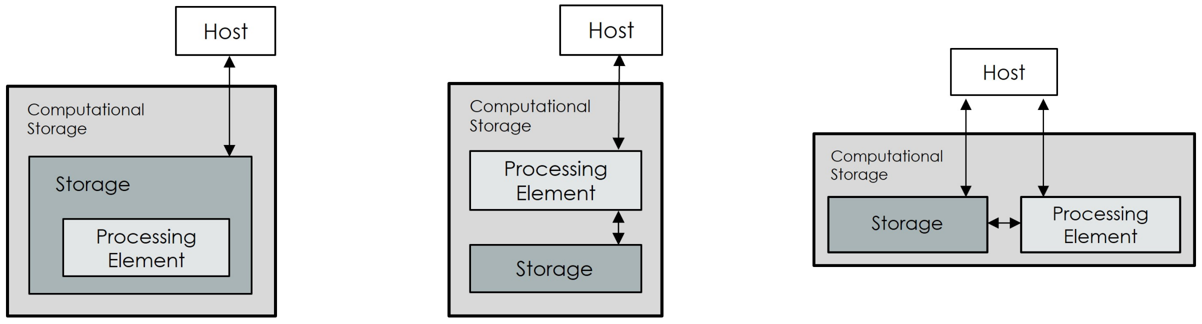
A wide variety of applications can benefit from smart storage but, in the context of this paper, we focus on their use in analytical databases. Among the platforms that we surveyed, many are designed specifically for use in databases and SQL operators, e.g., Ibex [27], AQUOMAN [28], YourSQL [15], FANS [19] and NASCENT [21]. Others are designed for general purpose offloading but are evaluated predominantly on database use-cases, e.g., Wisconsin Smart SSD [9], Biscuit [12], Summarizer [16], Insider [20], and NGD Newport [13].

It can be also seen in Table 1 that most of the selected related works have been evaluated, entirely or partially, using SQL operators. The datasets and workloads are typically generated using TPC-H. The size of the datasets used in the evaluation ranges from tens or hundreds of gigabytes (in most related work) to a terabyte (in AQUOMAN [28]). It is important to point out that most related work only considers a subset of TPC-H queries: they typically omit the queries that cannot benefit from in-storage acceleration. Even though this allows the authors to focus on studying the scenarios where smart storage shines, it makes it difficult to understand how database systems equipped with smart storage will behave on the entirety of a large workload, comprised of varied queries.

In addition to SQL, a subset of the related work [12, 13, 20] explores the benefits of offloading in other applications. They study operations such as data decompression, K-nearest neighbor search, and accelerate distributed ML training and inference tasks.

The studied related work exhibits a large variation when it comes to the type of the processing element used and can be grouped into ones that use low-power embedded CPUs [9, 13, 16], FPGAs or similar ASICs [19–21, 27], or a combination of the two [12, 15]. Additionally, we can categorize the related works into three categories, based on the location of the processing element relative to the flash storage device (also shown in Figure 1):

(a) **Shared processor / co-processor**: In early approaches to building smart storage, the embedded CPU of the SSD itself was used to perform computation [9]. Since such embedded CPUs have low performance and few cycles free for computation, there is also an option to deploy a larger CPU or a heterogeneous co-processor with it. The benefit of this architecture is that all data passing through the SSD control logic can be accessed by the offloaded operator without overhead – it requires, however, very



(a) Processing happening in CPU shared with SSD controller (co-processor)

(b) Dedicated processing element in the data path

(c) Dedicated processing element sits on the side of the storage

Figure 1: The processing in Smart Storage solutions is usually positioned in one of these three categories.

tight logical and physical integration of core SSD functions (such as FTL) and the offloaded computation.

(b) **In data path**: In this category, all of the data that is accessed by the host will pass through the processing element. In such architectures, the processing element needs to process the data at the same or higher bandwidth than that of data loading from storage to host – otherwise, the presence of the processing element would lead to slowdowns.

(c) **On the side**: In this third category, the processing element is connected to the storage device through an interconnect (e.g., PCIe), and both of them are connected to the host, either as a single logical unit (e.g., as the Samsung SmartSSD [2]) or as two separate ones (e.g., as the NGD Newport SSD [13]). Data has to be moved specifically for the purpose of processing from the storage medium to the processing element, before the results can be sent to the host. This means, on the one hand, that only the data pertinent to the offloaded task is loaded from the storage drive but, on the other hand, there is additional data movement happening in the system, introducing latency and potential bandwidth bottlenecks.

In the following, we look briefly at the details of the selected architectures: the overview of the system architecture, how they offload the computation to the storage, and what kind of processing element they use to run the offloaded computation.

Wisconsin Smart SSD [9]. Wisconsin Smart SSD is a framework that implements the core of the software that is needed for offloading and running user-defined applications in the smart storage. Its overall architecture is depicted in Figure 2. Embedded processors in the SSD are used to run the offloaded computations. This framework consists of two parts: 1) a communication protocol between the host and the smart storage, and 2) application programming interfaces (APIs) to be used in the user-defined programs that will run on the embedded processors.

This framework is evaluated using a SAS SSD with Microsoft SQL Server. Three SQL queries consisting of selection, projection, and aggregation are used to evaluate the framework (the framework is general and could also be used to run other queries or programs as well). To offload the computation to the storage, the user program must first start a session using the communication protocol's commands, and then it can use the protocol's GET command to retrieve the status or final results from the storage. With multiple session IDs, several programs can run in parallel inside the storage.

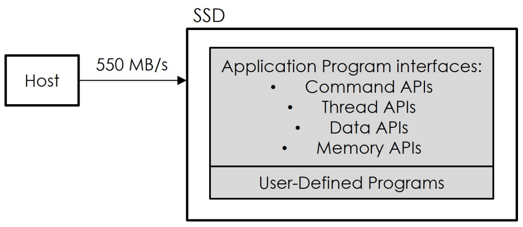


Figure 2: Wisconsin Smart SSD framework. A set of APIs is provided to be used in the offloaded user-defined programs.

Ibex [27]. Ibex uses an FPGA to design and implement an intelligent storage engine for the MySQL database. The FPGA is placed in the data-path between the storage and the host. Ibex is the only work with in-data-path processing that processes the data at line-rate. As it's shown in Figure 3, there are three main components implemented in the FPGA: 1) *Parsing and Projection*, in this component, the raw data coming from the storage media is parsed and based on the control information from the host, the database tuples are "narrowed" down to the columns that are part of the projection. 2) *Selection*, this component processes the data from the Parser component in a pipelined manner. For each tuple, one match signal is generated and it is set to high if the tuple matches the predicates in the WHERE clause. The Selection component supports the WHERE clauses consisting of predicates which are a comparison between a fixed-length column (e.g. integers) and a constant (it means that predicates such as comparing with a string should be executed in software). This component can support a certain number of predicates, and WHERE clauses containing more than this specific number could be evaluated both in software and hardware. 3) *Group-by Aggregation*, in this component, a specialized fully-pipelined hardware hash table is designed to handle group-by operations in queries. In case of occurring hash collisions, the tuples

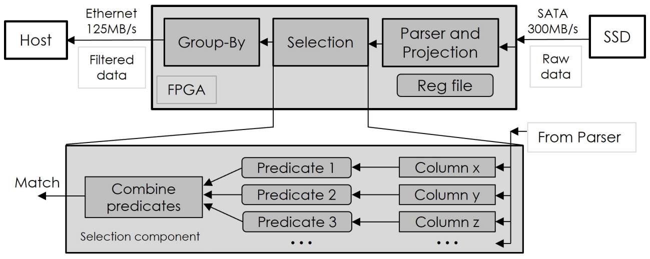


Figure 3: Overview of the offloaded operations in Ibex. The processing element (FPGA) is placed in the data-path and processes the data at line-rate.

will be bypassed to be run in the host software. To implement this hash table, it's possible to use both BRAM and DRAM. A write-through caching layer is created when using DRAM to prevent data hazards and mitigate the effect of DRAM read latency.

To extend the MySQL storage engine and to communicate with the FPGA, a driver is implemented and is accessible within the MySQL source code. Gigabit Ethernet is used for communication between the host and the FPGA, and a slow SATA II link allows the FPGA to access the SSD. As shown in Table 1, Ibex and Wisconsin Smart SSD are two old works that target slow storage protocols.

Biscuit [12] and YourSQL [15]. Biscuit is an in-storage computation framework that has been designed with a focus on being programmable and general. It uses the embedded processor of the SSD and also a hardware pattern matcher in each channel of the SSD to run the offloaded computations, which makes it the only related work that makes use of a heterogeneous design. The YourSQL database system, on the other hand, is integrated with Biscuit and is implemented on a branch of MariaDB. To use the Biscuit framework, YourSQL makes code changes to the query planner and the storage engine of the MariaDB. Both works use all TPC-H queries for evaluation.

The overall system architecture of Biscuit is shown in Figure 4. The programming model of Biscuit consists of computation units and coordination units. The offloaded tasks running inside the embedded CPU of the SSD are the computation units (SSDlets in Figure 4). The coordinator units are the tasks running in the host and are responsible for creating, managing, and also establishing the relation of producer and consumer among the tasks running in the embedded CPU. There are two libraries provided to write programs to be run in the SSD (libslet) and the host(libisisc). The host library has a proxy class that acts as an interface to the programs running inside the SSD (as shown in Figure 4).

In YourSQL (Figure 5), the query planner and the query executor are aware of the in-storage processing. The decision of whether to offload a query or not is taken automatically in the query planner. This is the only work that estimates the benefit of in-storage computation before offloading each query. To realize which table benefits more from offloading (this relates to join operations when there are multiple tables), YourSQL uses two metrics. The first one is the "Limiting score" which shows how restrictive the filter predicates are (for that specific table). The second one is the "filtering ratio" which is estimated using a "sampler task". YouSQL offloads

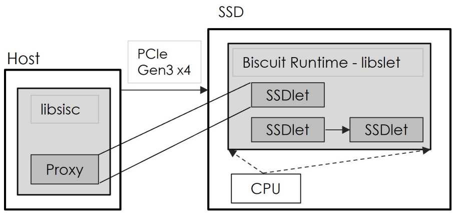


Figure 4: Biscuit system architecture. A library is provided in Biscuit to be used in the host system and acts as an interface for the tasks running in the SSD's processor.

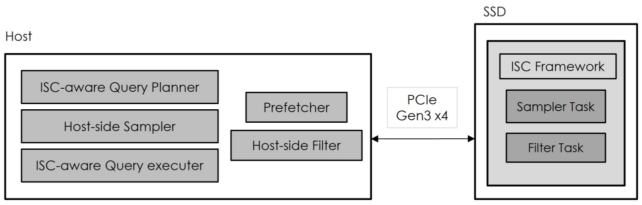


Figure 5: YourSQL system architecture. The decision to offload a task or not is made based on the task's potential benefit from offloading.

the sampler task to the SSD. It scans a small portion of the table instead of scanning all of it. Then, it estimates the "filtering ratio" which is calculated by dividing the number of match pages by the number of scanned pages by the sampler task. Considering one join query, YourSQL lists the tables with filter predicates, eliminates the small tables, selects a table with the highest Limiting Score, and estimates the Filtering Ratio for that table. If the estimated Filtering Ratio is higher than a threshold, then it offloads the filtering task for that specific table and places it first in the join order (to narrow the intermediate tables).

Summarizer [16]. In this framework, a set of APIs are designed to be used in the host applications in order to offload the tasks to the storage. The authors use the embedded processor of the SSD to run the user-defined tasks. Among the studied smart storage solutions, this is the only work that dynamically monitors the tasks that are waiting for in-storage processing and decides whether to run a task in storage or not. As it is shown in Figure 6, Summarizer consists of three main components that are implemented in the SSD's embedded processor: 1) Task Queue, which is a circular queue and contains pointers to the user functions that must be run when the host requests in-storage processing, 2) Task Controller, which decides whether to run a requested in-storage task or not, 3) User Functions Stacks, which store the user-defined functions. The Task Controller module works in two modes: Dynamic Mode and Static Mode. In Dynamic Mode, the controller makes the decision to run the requested in-storage task inside the SSD or not based on the empty slots in the Task Queue. Therefore, if there is not an empty slot in the Task Queue, the unprocessed data is transferred to the host even if the host requested in-storage processing. Conversely, the Static Mode always runs the requested in-storage processing task, regardless of the load on the embedded CPU.

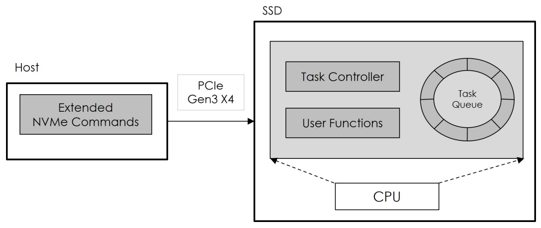


Figure 6: Summarizer high-level view. The Task Controller decides whether to run an offloaded task or not based on the number of waiting offloaded tasks in the Task Queue.

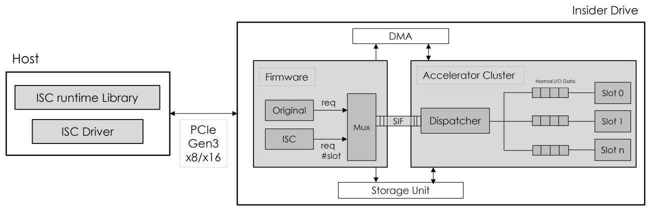


Figure 7: Architecture of Insider. An adaptive bandwidth scheduler is used to support multiple simultaneous applications running in the drive.

To evaluate the smart storage system, data integration algorithms and three TPC-H queries are offloaded to the storage. The APIs to be used for offloading the computation are implemented by modifying the NVMe interface between the host and storage and consist in new NVMe commands to initialize, compute and finalize the in-storage computations. The purpose of the initialization command (INIT_TSKn) is to inform the Task Controller that the host wants to run task "n" (n is used as task ID) inside the storage and to initialize variables. Two commands (READ_PROC_TSKn and READ_FILT_TSKn) are used to send the desired task ID to be performed after reading the data from flash. After data is fetched from flash, the Task Controller decides to run the task for the loaded data or not. If it decides to execute the computation in storage, the desired user function is invoked using the pointer in Task Queue. In the end, the host can collect the result from storage using the finalizing command (FINAL_TSKn).

Insider [20]. This storage system is the only general platform implemented using an FPGA that also employs a flow control technique for sending data between the host and FPGA. As shown in Figure 7, it consists of two main components that are inserted inside the storage: 1) an Accelerator Cluster, which is responsible for running the user-defined offloaded tasks and is designed using an FPGA. It consists of multiple slots and using partial reconfiguration, new accelerators could be loaded to a particular slot by host users. 2) the Firmware, which is responsible for interacting with the host-side in-storage computing (ISC) runtime library and driver. This component is made by an extension to the original drive firmware. By separating these two components, the control and data plane are separated; the accelerator cluster is in the data plane and does not decide where to read/write data from/to. By separating the accelerator cluster from the control plane, user-defined program execution is isolated from the system component in the control plane. A virtual file abstraction, that could be accessed via a subset of POSIX-like I/O APIs, is provided as the programming model of the host side. In addition, a host-side library is implemented which is available from the user-space and cooperates with the drive hardware to support the flow control.

As there are multiple slots in the FPGA, parallel execution of multiple tasks is possible. To support sending drive data to various slots, a multiplexer and an ISC unit are inserted into the firmware. Anytime the ISC unit receives a request from the host (with logical block address and the slot index), it forwards the request to the storage unit and pushes the slot index into a FIFO (Slots Index FIFO, i.e. SIF in Figure 7). Then the dispatcher forwards the data from storage to the popped slot index from SIF. Slot 0 is reserved for normal I/O requests (i.e. without ISC) and contains pass-through logic. There is also a dynamic weighted policy implemented to divide the drive bandwidth fairly between various application slots.

AQUOMAN [28]. AQUOMAN (Analytics QUery Offloading MAchiNe) is an FPGA-based architecture that proposes an end-to-end system for in-storage query acceleration. This general analytic query processor is the only proposed microarchitecture for running queries, including multi-way joins. The proposed architecture aims to speed up column-store databases and is integrated with the MonetDB [1] column-store database. The AQUOMAN and the host can access the NAND flash memory at the same time. To run a query in this accelerator, each query must be compiled into a set of "Table Tasks" and the accelerator's components are configured based on the parameters of the Table Tasks.

As shown in Figure 8, There are three main accelerators in AQUOMAN's Architecture. 1) Row Selector, this component is responsible for evaluating the predicates and will send a bit vector mask showing the qualified selected rows to the next component. 2) Row Transformer, this component reads the data table from NAND flash based on the bit vector mask (i.e. only reads the pages including qualified rows) and consists of several processing elements to apply computation on each row and produce the intermediate table to send to the "SQL Swissknife" component. The processing elements are initialized based on the Table Task, i.e. Table Task defines the instructions to be performed on data. 3) SQL Swissknife, this component has direct access to AQUOMAN's DRAM and consists of an array of accelerators to run standard SQL operators (e.g. accumulate, merge,...). There are seven operators implemented in this component and new operators can be added. The connection between the input of the component and the accelerators is configured based on "Table Tasks". The intermediate tables are stored in the DRAM and retrieved again later for merging.

NASCENT [21] and FANS [19]. NASCENT and FANS are FPGA-accelerated near-storage solutions that leverage Samsung SmartSSD [2]. Samsung SmartSSD is an NVMe flash drive that has an additional FPGA accelerator and FPGA DRAM. The host processor can initiate FPGA DRAM read/write requests and also send computation requests to the FPGA. Furthermore, the host can initiate peer-to-peer (P2P) connections between the FPGA DRAM and the NVMe SSD. Once a connection is established, the host is no longer involved and the data is transferred using the PCIe P2P link. In the related

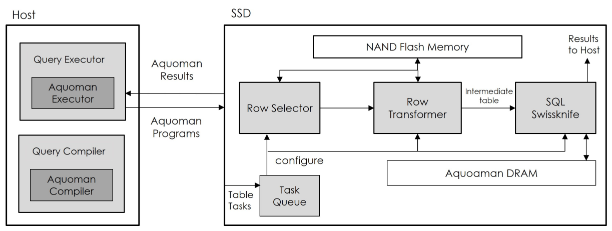


Figure 8: AQUOMAN proposes a microarchitecture for running queries, including multi-way joins.

work, we found one other instance of FPGA-based smart storage drive, Scaleflux [3, 7], that uses the FPGA to run both flash memory control and offloaded computation. For running other functions such as address mapping and garbage collection, the Scaleflux drive is managed by the host.

As shown in Figure 9, the NASCENT accelerator is made up of three modules. The first one is the dictionary decoding kernel, which decodes the stored data to provide the input for the sort module. The second module, a sort kernel, implements bitonic sort and can be scaled based on the FPGA resources. The last module, a shuffling kernel, is in charge of reordering the rest of the table based on the sorted key column. To improve the access patterns for both the sort kernel and the shuffle kernel, one particular storage format is proposed that stores the key column in a column-store format and the rest of the table as row-store (the sort kernel benefits from column-store since it needs to access the key column and the shuffle kernel benefits from row-store as it needs to access elements in one row to change their location). Since the shuffling module requires random reads from memory to access the table rows, multiple shuffling kernels are used in the NASCENT architecture to fully utilize the bandwidth available between the storage and FPGA. Furthermore, a single dictionary decoder module is instantiated and the rest of the FPGA resources are used for the sorting module. The sort kernel uses FPGA BRAM to store the data and if the sorting elements do not fit in the BRAM blocks of the sort module, the sort kernel first sorts a chunk of the input, writes the result back to the DRAM, sorts the next chunk of data, and eventually merges the sorted data chunks stored in the DRAM.

The FANS accelerator design divides the sorting process into two phases: the sorting phase and the merging phase. The FPGA needs to be re-programmed to transition between these two phases, which are both implemented using a merge tree with various tree configurations. In the sorting phase, after sorting a chunk of the data, the sorted data is written back to the DRAM and then the next chunk of the data is sorted. The sorted data chunks are then merged during the merging phase. The size of the merged data is usually bigger than the FPGA DRAM capacity and the partially merged data should be written back to the flash in this case.

NGD Newport [13]. In this smart storage drive, a dedicated processor, running at 1GHz, is inserted inside the SSD to run the offloaded applications and can run a full operating system (e.g. Linux). This is the only commercially available system among the studied works that inserts a dedicated CPU with the SSD controller on the same custom chip to run the offloaded computations. As shown in Figure 10, there are three components in the design of this drive: 1) the front-end subsystem, which is responsible for the communication between the host and the drive, and notifies the back-end subsystem if a new I/O command is received. 2) the back-end subsystem, which is responsible for managing the NAND flash memory and tasks such as address translation and garbage collection. 3) the in-storage processing (ISP) engine, which is responsible for running offloaded applications.

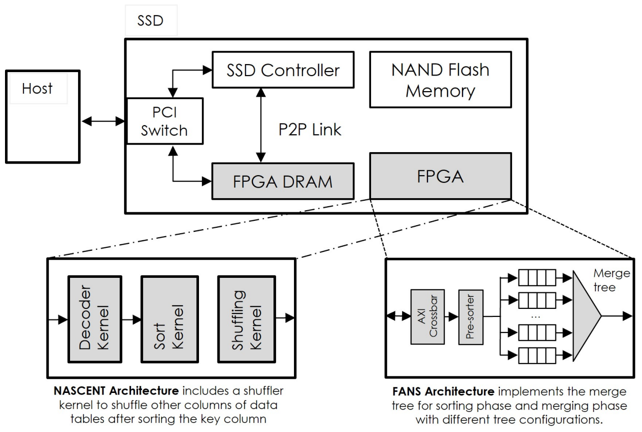

SSD
Host
SSD Controller
NAND Flash Memory
PCI Switch
P2P Link
FPGA DRAM
FPGA
Decoder Kernel
Sort Kernel
Shuffling Kernel
AXI Crossbar
Pre-sorter
Merge tree
NASCENT Architecture includes a shuffler kernel to shuffle other columns of data tables after sorting the key column
FANS Architecture implements the merge tree for sorting phase and merging phase with different tree configurations.

Figure 9: NASCENT and FANS system architecture. Both systems use Samsung SmartSSD and the difference is in the designed FPGA accelerator.

There is also a TCP/IP tunneling system implemented on the regular PCIe between the host and the dedicated processor for in-storage processing (ISP). Using this link, the host programs can easily communicate with the applications running in the ISP with a TCP/IP link. Using this link, the dedicated processor inside the SSD could be also used as a node in a distributed application.

A Customized Block Device Driver (CBDD) is implemented to enable the ISP to access the flash memory (by communicating with the back-end system). This driver is optimized for on-chip communication. Another important feature of the Newport device is the shared file system between the host and ISP. To provide synchronization between the ISP engine and the host's operating system, a software layer has been implemented. This synchronization technique makes it possible for the operating system in the host and the ISP engine to concurrently mount one storage media.

3 PRINCIPLES FOR DESIGNING FUTURE SMART STORAGE

Studying recent related works reveals that the choice of data layout, workload characteristics (such as selectivity of query mix) as well as the level of integration of the hardware accelerator result in widely different performance improvements. Analyzing these factors teaches us lessons that will be useful when designing future smart storage devices or when determining which types of operations of novel data-intensive applications to offload to the storage. In the following, we present the three main "guiding principles" we identified and provide supporting evidence from related work.

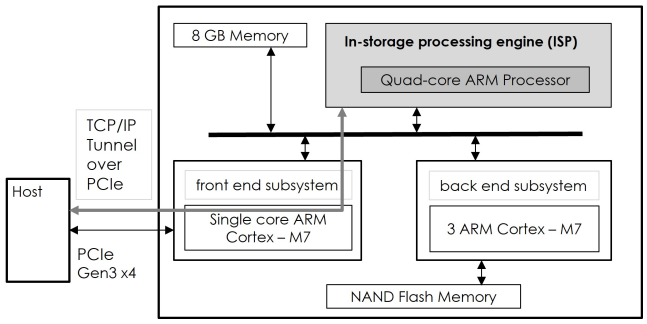


Figure 10: NGD Newport architecture. It employs a dedicated general processor (ISP) for running offloaded computations.

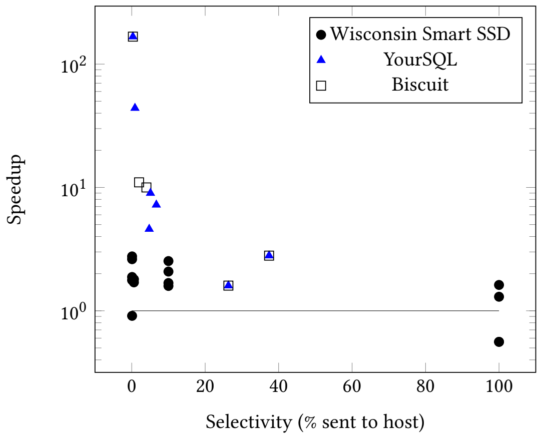


Figure 11: Relative execution time for various selectivity numbers. The line in the figure represents a value of one. Therefore, dots above the line indicates system speedup, while those below shows system slowdown.

3.1 Balance the compute cost of offload against the data selectivity

When offloading a task to smart storage, a significant reduction in data movement does not always lead to end-to-end performance improvement; one must also consider the compute requirements of the offloaded task and the compute capacity of the processing element in the storage.

Most related work focuses on a relatively narrow set of filtering operations (e.g., equality filtering, string search) that typically only bring a fraction of the data up to the host. Hence, the data movement bottleneck can be greatly reduced. Figure 11 depicts some of the speedup numbers reported in the studied related works for various selectivity values [1]: as a rule of thumb, higher selectivity (less tuples selected and sent to host) leads to higher speedup and the low selectivity cases (most tuples sent to host) result in a slowdown or only a minor speedup. Herein lies a challenge: in order to decide what filter to offload, the system needs to predict the selectivity of the query – this can be difficult to always get right.

Even with precise selectivity estimation, query speedup cannot always be predicted. As an illustrative example, consider that even though we would expect a large speedup with highly selective query, Figure 11 shows that there is a case for Wisconsin Smart SSD where the system slows down even with 0.1% selectivity (relative speedup of 0.91x). The reason for this is that the storage's embedded CPU becomes a bottleneck.

As an alternative to selectivity estimation and cost functions, in some cases it is possible to design the processing element such that it operates at a fixed rate regardless of the complexity of the offloaded operations (granted, of course, that this operation "fits"). For instance, in Ibex [27], filtering and group by offloading always processes the data at constant rate and, even if the offloading does not result in data reduction, the overall system does not experience slowdowns.

[1]It should be noted that the numbers were calculated in different experiments and are shown in one graph to demonstrate how selectivity affects the speedup.

3.2 Matching the data layout with the expected layout of the application and processing element

In order to get the most out of specialized hardware, the data layout adopted by the database should ideally match what the processing element handles most efficiently, Otherwise, performance benefits will be significantly reduced due to additional data transformations.

A concrete example of how data layout transformations cause overhead can be found in the recent work by Sidler et al. [24], that builds an FPGA-based regular expression matching accelerator for databases. The FPGA is deployed not in storage but as a co-processor in an Intel Xeon+FPGA machine. In this system, the data is passed to the accelerator from the database in PAX-like pages. These need to be parsed and realigned before filtering can take place. The time required for this data transformation is roughly half of the time required for running the main computation – even though, in this particular design, these operations are fully pipelined, a significant portion of the FPGA is taken up by the parsing logic. With a better matching data layout, these FPGA resources could be used to increase the regular expression matching circuit capacity.

Generally, avoiding loading data that is not related to the offload to the processing element will be beneficial. For instance, as shown in Table 2, the Wisconsin Smart SSD architecture shows better performance improvement with the PAX layout (in which one column of the data table is grouped together inside a page) [6] than with the N-ary storage model (NSM) layout (row-store). The reason for this is that the smart SDD's processor has an instruction that enables it to load multiple data from the internal DRAM of the device at once. As a result, with the PAX layout, only the column values required for calculating the predicates of interest are loaded. This increases DRAM access bandwidth and reduces runtime.

The NASCENT architecture is an other example that adopts a custom data layout that benefits the accelerator. The sort kernel benefits from column-store as it needs to access the key column elements, whereas the shuffle kernel benefits more from row-store

Table 2: Speedup for Various Data Layouts [9]

Query	NSM (row-based)	PAX (col-based)
`select, selectivity=0.1%`	1.77	2.63
`select-aggr., selectivity=0.1%`	1.88	2.76

since it needs to access the element of one row together. As a result, NASCENT stores the key-column consecutively and the rest of the data table as row-store. In this way, data could be loaded sequentially into both kernels.

3.3 Embrace heterogeneity

Most related works rely on either a low-power CPU or a specialized FPGA or ASIC. The reason for this is often that choosing one or the other simplifies the complexity of the prototype that needs to be built. Sometimes, authors rely on an FPGA exclusively to demonstrate that such devices can be used to build complete functional systems – even though the addition of small CPU cores to the reprogrammable fabric might be a better engineering decision. After reviewing the related work, for future systems we strongly advocate for a heterogeneous design. It allows for "best of both worlds", running parts of the offloading logic that benefit from iterative processing on CPU cores, and parts that benefit from massive parallel execution and pipelining on specialized FPGA/ASIC cores. There are examples of systems adopting such an approach. For instance, Biscuit [12] and Catalina Smart SSD [26] both rely on specialized accelerators cores in addition to code running on the CPU that is the SSD controller.

Having a heterogeneous architecture, along with paying attention to designing the architecture in such a way that allows running more than one type of processing at the same time to fully utilize the resources, makes the smart storage more useful for a broader range of applications. However, finding the right operations to design the specialized hardware for, as well as circuits that are not too specialized for one specific operation but can handle running multiple operations in the smart storage, requires more attention.

Figure 12 shows the speedup of running a simple string search using the Biscuit [12] system. To run this application, Biscuit uses its hardware pattern matchers which are placed in each channel of the SSD. These pattern matchers give Biscuit a high search throughput and the ability to run the task in parallel. Using these parallel resources, increasing the system's load has little impact on Biscuit's execution time (in the host system, the execution time increases when the system's load is increasing and as a result, the speedup is increased as shown in Figure 12). However, if Biscuit would use only its embedded CPU, which is low-power and slower than the host CPU, to run this offloaded task the execution time would be different. It would evolve in a similar manner to the host's execution time (that is, with increased parallelism, various overheads would be incurred). As a result, using only the embedded CPU, without hardware accelerators, would reduce the achievable speedup in this application.

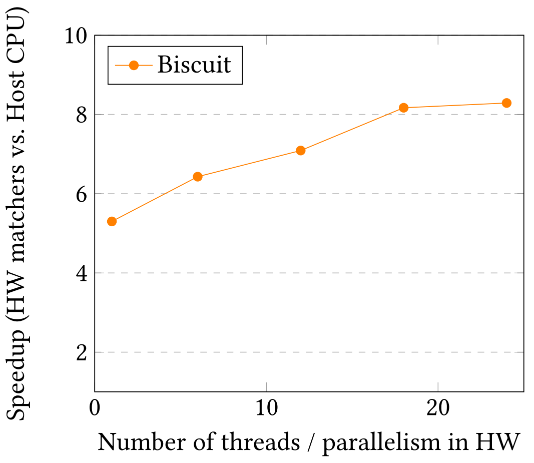


Figure 12: In Biscuit [12], the hardware pattern matchers are more efficient for highly parallel workloads than the host CPU. Therefore, the achievable speedup increases slightly with the parallel load on the system

4 CLOSING THOUGHTS

In this brief survey, we looked at the lessons and opportunities from a large number of smart storage solutions from the last decade and identified three principles that could be applied in the design of future smart storage devices. First, highly selective queries and reduced data movement do not "automatically" guarantee performance improvements, and when deciding whether to offload or not to offload a task, one needs to take into account the compute capacity of the smart storage as well as the compute requirements of the operators. Second, to prevent wasting processing resources for unnecessary data transmissions, it is important to ensure that the data layout matches the expected layout of the processing element and application. Third, there are both efficiency and flexibility arguments for using a heterogeneous architecture instead of basing smart storage purely on low-power CPU cores or fully specialized ASIC/FPGA units.

Industry and research trends indicate that the potential of using smart storage has been recognized and they are no longer "exotic" devices. At the same time, various issues still need to be addressed in the design of smart storage devices. In our opinion, the following are worthwhile directions for future research: designing tightly couples heterogeneous offloading architectures (e.g. FPGAs alongside CPUs, with fine-grained work partitioning), providing efficient scheduling and resource management mechanisms, and enabling low-overhead performance isolation between different applications using the same smart storage device.

ACKNOWLEDGEMENT

This work has been partially funded by a grant from the Huawei Munich Research Center.

REFERENCES

[1] 2022. "MonetDB". https://monetdb.org/.

[2] 2022. "Samsung SmartSSD". https://semiconductor.samsung.com/ssd/.

[3] 2022. "Scaleflux". https://www.scaleflux.com.

[4] 2022. "Sort Benchmark". http://www.ordinal.com/gensort.html.

[5] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review* 32, 5 (1998), 81–91.

[6] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} Meets Computational Storage: Efficiently Support Analytical Workloads in {Cloud-Native} Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 29–41.

[8] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. 2013. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 91–102.

[9] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[10] Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Arvind Arasu, and Donald Kossmann. 2019. Blockchaindb-towards a shared database on blockchains. In *Proceedings of the 2019 International Conference on Management of Data*. 1905–1908.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 365–376.

[12] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 153–165.

[13] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. [n.d.]. Leveraging Computational Storage for Power-Efficient Distributed Data Analytics. *ACM Transactions on Embedded Computing Systems (TECS)* ([n. d.]).

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7 (2019).

[15] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[16] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 219–231.

[17] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M Patel, Christopher Ré, and Michael Stonebraker. 2021. ML-In-Databases: Assessment and Prognosis. *IEEE Data Eng. Bull.* 44, 1 (2021), 3–10.

[18] Mark Oskin, Frederic T Chong, and Timothy Sherwood. 1998. Active pages: A computation model for intelligent memory. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 192–203.

[19] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-Accelerated Near-Storage Sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 106–114.

[20] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. {INSIDER}: Designing {In-Storage} Computing System for Emerging {High-Performance} Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 379–394.

[21] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-storage acceleration of database sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 262–272.

[22] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 67–80.

[23] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2019. Understanding and Benchmarking the Impact of GDPR on Database Systems. *Proceedings of the VLDB Endowment* 13, 7 (2019).

[24] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 403–415.

[25] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active Flash: Towards {Energy-Efficient},{In-Situ} Data Analytics on {Extreme-Scale} Machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 119–132.

[26] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Catalina: in-storage processing acceleration for scalable big data analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 430–437.

[27] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974.

[28] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. Aquoman: An analytic-query offloading machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 386–399.