Adaptive Recovery for SCM-Enabled Databases

Ismail Oukid[†‡] Anisoara Nica[‡] Daniel Dos Santos Bossle[◇*]

Wolfgang Lehner[†] Peter Bumbulis[‡] Thomas Willhalm[+]

[†]TU Dresden first.last@tu-dresden.de
[‡]SAP SE first.last@sap.com
[+]Intel Deutschland GmbH first.last@intel.com
[◇]Grenoble INP - Ensimag first.last@grenoble-inp.org

ABSTRACT

Storage Class Memory (SCM) has the potential to drastically change the database system landscape – much like high core count CPUs and large DRAM capacities spurred a shift to in-memory databases a decade ago. One of the possibilities provided by SCM is to significantly improve restart performance. SCM-enabled databases can evince a single-level main-memory architecture that stores, accesses, and modifies data directly in SCM, removing the traditional recovery bottleneck for main-memory databases: reloading data from durable media to main memory. Almost instantaneous recovery is possible, but at the cost of reduced throughput as latency sensitive data structures such as indexes need to be kept on SCM whose read and write latencies are projected to be noticeably slower than those of DRAM. We can regain this throughput by fully storing secondary data structures in DRAM and rebuilding them after restart, concurrently with processing incoming requests. While these data structures are being rebuilt, request throughput will be reduced not only because of the rebuilding overhead but, more significantly, because their (temporary) absence will result in sub-optimal access plans. Hence, rebuilding DRAM-based data structures becomes the new bottleneck for SCM-enabled databases recovery. In this paper, we address this bottleneck and describe *Adaptive Recovery*, a novel recovery technique that significantly reduces the impact on request throughput during the recovery period over naïve approaches such as rebuilding data structures when first referenced.

1. INTRODUCTION

Availability guarantees form an important part of many service level agreements (SLAs) for production database systems [7]. Database recovery time has a significant and direct impact on database availability as many database crash conditions are transient (e.g., software bugs, hardware faults, and user errors) and for these, restarting is a reasonable approach to recovery. To ensure transaction durability, traditional in-memory DBMSs periodically persist a copy (checkpoint) of the database state and log subsequent updates. Recovery consists of reloading the most recent persisted state, applying subsequent updates, and undoing the effects of unfinished transactions. For in-memory DBMSs reloading the persisted state is typically the bottleneck in this process.

The advent of Storage Class Memory[1] (SCM) has empowered a new class of database architectures where memory and storage are merged [1, 29, 23, 37]. SCM-enabled database systems keep a single copy of the data that is stored, accessed, and modified directly in SCM. This eliminates the need to reload a consistent state from durable media to memory upon recovery. SOFORT is our own SCM-enabled database system. It is a prototype hybrid SCM-DRAM in-memory storage engine that allows secondary data structures (indexes, materialized views, etc.) to be kept either in DRAM or in SCM. While SOFORT can achieve near instant recovery if most secondary data structures are stored in SCM, there is a query performance cost, as many of these data structures are latency-sensitive and SCM is expected to have higher latencies than DRAM. In previous work [31], we explored trading off query performance for recovery performance by judiciously placing certain secondary data structures in SCM. In this paper however, we assume that all latency-sensitive secondary data structures are kept in DRAM as our goal is to improve recovery performance without compromising query performance.

SOFORT provides two different recovery strategies. In *Synchronous Recovery* [29] on restart, after recovering the SCM-based data structures, SOFORT rebuilds the DRAM-based secondary data structures and then starts accepting requests. If the DRAM-based secondary data structures are large, restart times can still be unacceptably long. To address this, we devised an *Instant Recovery* strategy [31]. It allows queries to be processed concurrently with the rebuilding of the DRAM-based data structures. However, while the secondary data structures are being rebuilt, request throughput is reduced. Part of the performance drop is due to the overhead of rebuilding but a more significant factor is the unavailability of the DRAM-based secondary data structures, resulting in suboptimal access plans.

In this paper we propose a novel recovery strategy, *Adaptive Recovery*, that can benefit any main memory database, but is most relevant for SCM-enabled databases, as they alleviate the traditional recovery bottleneck of reloading data from storage to main memory, making rebuilding secondary data the new recovery bottleneck. Adaptive recovery is inspired by the observation that not all secondary data structures are equally important to a given workload. Adaptive recovery improves on instant recovery in two ways. First, it prioritizes the rebuilding of DRAM-based secondary data structures based on their benefit to a workload (instant recovery simply uses an arbitrary order). Second, it releases most of the CPU resources dedicated to recovery once all of the important secondary data structures have been rebuilt (instant recovery statically splits CPU resources between recovery and query processing for the entire recovery period).

Adaptive recovery aims at approaching peak performance of the database as fast as possible by tuning the rebuild order of secondary data structures to optimize the workload run concurrently with the recovery process. To determine the optimal rebuild order, we conduct a characterization of secondary data structures in the SCM era. Although we focus on indexes in this work, the characterization

[*]This author contributed to this work while interning at SAP.

[1]SCM is also referred to as *Non-Volatile RAM* (NVRAM) , *Persistent Memory*, or simply *Non-Volatile Memory* (NVM)

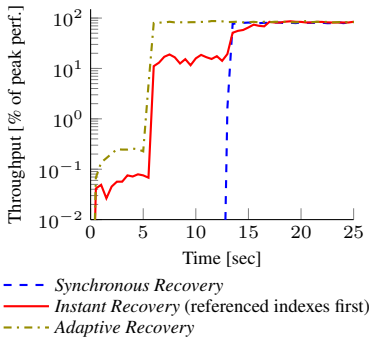


Figure 1: Results overview: TPCC transaction throughput during recovery.

and the recovery algorithms are generic and apply to most runtime data structures. We identify a set of properties that are intrinsic to secondary data structures and infer from them the criteria based on which they should be ranked, namely: (1) the usefulness to the workload before failure and during recovery, (2) the cost of rebuilding each secondary data structure, and (3) workload-related dependencies between secondary data structures.

We propose two ranking functions that take into account these properties. The first one satisfies items (1) and (2) while the second one satisfies all three of them. Additionally, the ranking of secondary data structures is adjusted dynamically to accommodate workload changes during the recovery period. Furthermore, we introduce a resource allocation algorithm that releases recovery resources to query processing when it detects that all important secondary data structures have been rebuilt.

We conduct a detailed experimental evaluation that shows the benefits of our approach. Figure 1 shows potential performance results. Our adaptive recovery approach significantly outperforms synchronous recovery, since in adaptive recovery, the database approaches its peak performance well before the end of the recovery process: approaching peak performance is no more a function of the size of all secondary data structures but only of a small subset of them, namely the relevant ones to the current workload. An interesting observation is that synchronous recovery outperforms instant (recover referenced indexes first) recovery. This result proves that a sub-optimal ranking of secondary data structures can be harmful to recovery performance.

The rest of the paper is structured as follows: Section 2 gives an overview of necessary background. Then, Section 3 presents a characterization of secondary data structures and our adaptive recovery technique. Thereafter, we implement our approach in SOFORT and conduct a thorough experimental evaluation in Section 4. Finally, Section 5 surveys related work and Section 6 concludes the paper.

2. BACKGROUND

This section presents the context and the motivation of the research question we address in this paper. First, we briefly describe SCM and our hardware assumptions. Thereafter, we give an overview of our prototype SOFORT. Finally, we discuss existing recovery techniques for SCM-based main-memory databases.

2.1 Storage Class Memory

SCM is an emerging memory technology that combines the low latency and byte-addressability of DRAM with the non-volatility, density, and economic characteristics of existing storage media (HDDs, SSDs). SCM exhibits asymmetric latencies in the same range as those of DRAM, with writes noticeably slower than reads. Table 1 summarizes current characteristics of Phase Change Memory (PCM) [24] and Resistive RAM (RRAM) [14], two SCM candidates, based on the latest research and industry announcements, and compares them with current memory technologies. The table shows that PCM and RRAM have a latency profile in the range 200-300ns. Other promising SCM candidates that were subject to industry announcements include 3D XPoint [2], Spin Transfer Torque RAM (STT-MRAM) [17], and Memristors [43].

Table 1: Memory technologies comparison (adapted from [9]).

Parameter	DRAM	NAND	RRAM	PCM
Read Latency	60 ns	25 µs	200-300 ns	200-300 ns
Write Speed	~1 GB/s	2.4 MB/s	~140 MB/s	~100 MB/s
Endurance	10^{16}	10^4	10^6	$10^6 - 10^8$
Density	$1\times$	$4\times$	$2-4\times$	$2-4\times$

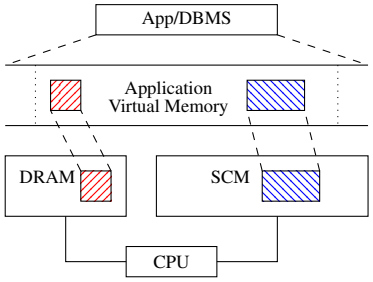


Figure 2: SCM is mapped directly in the address space of the application.

Similar to flash memory, SCM supports a limited number of writes. However, SCM promises to be more resilient; for example, PCM is two to four orders of magnitude more resilient than flash memory. Several works, such as [35], have already addressed this issue and proposed wear-leveling techniques to increase the lifetime of SCM.

SCM can be architected in different ways: as high-performance storage, as DRAM replacement, and as universal memory (main memory and storage at the same time). In the latter, SCM is either standalone or next to DRAM. The following sub-section presents our memory architecture assumptions.

2.2 Memory Paradigm

In this paper we assume that future systems will be equipped with a combination of DRAM and SCM that is visible to and controlled by the application layer, as illustrated in Figure 2. Besides, we consider that the latency of SCM will be in the range of $2-5\times$ the latency of DRAM.

Furthermore, we expect that SCM might be attached to processors in a way that allows memory semantics, i.e. the application can directly access SCM using existing load and store instructions. The cache hierarchy of today's processors might cause a problem for proper usage of SCM, as modified cache lines can still reside in cache when a power failure occurs. In this work, we use cache line flushing instructions and memory fences to enforce data durability.

SCM encompasses characteristics of both memory and storage. In contrast to normal volatile memory, the memory management of SCM requires means to rediscover memory after a restart. We follow the approach of [9] where a file-system-like interface allows to map SCM regions directly in the address space of the application.

[2] https://en.wikipedia.org/wiki/3D_XPoint

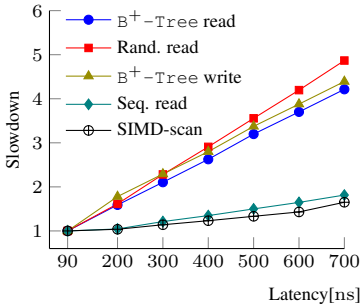


Figure 3: Slowdown incurred by using SCM over DRAM in different micro-benchmarks.

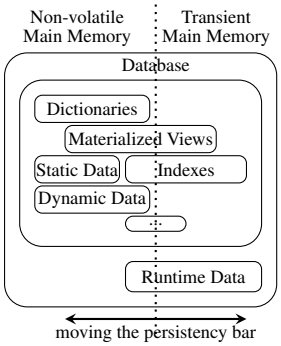


Figure 4: SCM-Enabled database architecture.

Basically, the application first needs to open a file and then map it using the existing *mmap* interface, acquiring direct access to SCM. Indeed, using the new *Direct Access*[3] (DAX) feature of the *ext4* file system, no additional copy is needed and modifications of the memory are directly persisted without any additional system calls. In contrast, the usual implementation of *mmap* copies data from SCM to DRAM.

SCM Emulation

To conduct our tests, we use a special system setup based on DRAM that provides two additional capabilities: first, part of the memory is separated out as a special memory region. We treat this memory region as *persistent memory*. Second, thanks to a special BIOS, the latency of this memory region can be configured. This allows us to simulate the latencies of next-generation SCM. While this SCM evaluation platform is equipped with two Intel Xeon E5 processors, we bind the application to the first socket during our experiments to uncouple NUMA effects from the effects of SCM's higher latencies. All tests were therefore conducted on a single processor with 8 cores, each running at 2.6GHz and featuring 32KB L1 data and 32KB L1 instruction cache as well as 256KB L2 cache. The 8 cores share a 20MB last-level cache. A full description of this emulation system can be found in [8].

2.3 Micro-benchmarks

To understand the performance implications of a hybrid SCM-DRAM hardware architecture, we designed the following micro-benchmarks: First, we evaluate the performance of sequential and random reads in DRAM and in SCM. The size of the dataset is 1 GB. Sequential read is implemented as a cache ping, i.e., one byte is read from every cache-line-sized piece of the dataset, making the whole of it go through L1 cache. As for random read, one byte from the dataset is randomly read until the amount of read data is equal to the size of the dataset. Second, as a real-world example of sequential reads, we evaluate the *SIMD-scan* [42], an OLAP scan operator over bit-packed integers usually representing dictionary-coded values in a columnar representation, in SCM and in DRAM. In this experiment, each integer value is represented with 11 bits, and the size of the dataset is set to 200 million values, i.e., ~262 MB. Last, as a real-world example of random reads, we evaluate read and write performance of a B+-Tree in DRAM and in SCM. In each experiment, we first warmup the B+-Tree with 10 million tuples, then we execute 10 million either read or write operations. Each tuple is a pair of 8-Byte integers.

In all experiments, we vary the latency of SCM from 90 ns (i.e., the latency of DRAM) up to 700 ns. Figure 3 depicts the experimental results. The y-axis represents the slowdown incurred by using SCM over DRAM. For an SCM latency of 200 ns, we notice that the slowdown for sequential read and SIMD-Scan is only 1.05× and 1.04× respectively, while for an SCM latency of 700 ns the slowdown increases but is limited to 1.81× and 1.65×, respectively. This is explained by hardware and software prefetching that hide the higher latency of SCM when detecting a sequential memory access pattern. Besides, the slowdown with higher SCM latencies is due to the fact that the hardware prefetcher is calibrated in the emulator for DRAM's latency. This can be fixed in real hardware by calibrating the hardware prefetcher for SCM's latency. We conclude that workloads with sequential memory access patterns do not (or hardly) suffer from the higher latency of SCM.

On the other hand, the performance of random read, B+-Tree read, and B+-Tree write quickly and linearly deteriorates as we increase the latency of SCM: for an SCM latency of 200 ns, the slowdown is already 1.61×, 1.59×, and 1.78×, respectively, and increases for an SCM latency of 700 ns up to 4.86×, 4.21×, and 4.40×, respectively. This is explained by the fact that a B+-Tree has a random access pattern that triggers several cache misses per read/write, amplifying the penalty provoked by the higher latency of SCM. We conclude that workloads with random access patterns significantly suffer from the higher latency of SCM. This result motivates the need to use DRAM to enable faster query execution, as keeping latency-sensitive data structures in SCM leads to a significant performance penalty.

2.4 SOFORT

The advent of new hardware technologies and architectures, such as SCM, is driving a necessary rethink of existing database architectures to leverage the new opportunities brought by advancement in hardware technology [38]. In that context, we explored the implications of SCM on database architectures and proposed SOFORT, a hybrid SCM-DRAM storage engine designed from scratch to harness the full potential of SCM [29]. SOFORT is a single-level store, i.e., it keeps a single copy of the primary data in SCM, directly operates on it, and persists changes in-place in small increments. In the following, we give a brief overview of SOFORT.

Architecture

Figure 4 outlines the move from a traditional database architecture towards a system design exploiting SCM in addition to traditional, transient RAM. In the traditional architecture, runtime data, the buffer pool, and the log pool are associated with devices for durability. All database objects in the buffer pool reflect a (potentially modified) *copy* of the database state; runtime information is stored in main memory and re-built during system startup. The demarcation line

[3] https://www.kernel.org/doc/Documentation/filesystems/dax.txt

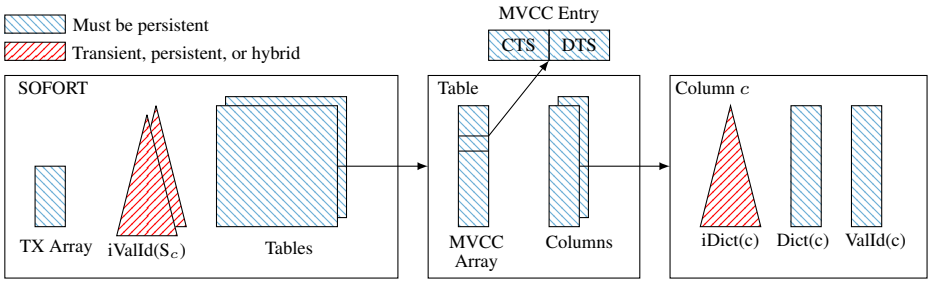


Figure 5: Overview of a column layout in SOFORT.

between transient memory and persistent storage is explicit and traffic is managed by the database system using an eviction policy, a log buffer, etc. Using a hybrid memory model consisting of traditional RAM on the one side and SCM on the other side allows for a radical change and a novel system architecture in different perspectives. First, there is no need to copy data out of the storage system but instead, we can directly modify the data stored in SCM. Secondly, the use of persistent data structures in combination with a compatible concurrency control scheme (e.g. MVCC) allows for completely replacing the traditional logging infrastructure by fine-grained, cheap micro-logging at data structures level. Lastly, certain database objects can be stored in transient RAM or in SCM as long as *information loss* can be guaranteed not to happen, i.e., index structures can be stored in transient RAM and reconstructed during recovery – more details can be found in [31]. Contrary to classical designs, system runtime information (e.g. transaction tables, etc.) can be easily and efficiently stored in SCM to improve system startup time. Hence, the new major bottleneck in such systems during recovery is the rebuild of DRAM-based secondary data structures. We address this bottleneck in this work.

Data layout

SOFORT is a column-oriented in-memory storage engine that uses dictionary compression. Figure 5 gives an overview of data organization in SOFORT. Tables are stored as a collection of append-only columns. Each column c consists of an unsorted dictionary $Dict(c)[]$, stored as an array of the column's unique data values, and an array of value ids $ValId(c)[]$, where a value id is simply a dictionary index (position). For a given column c, $Dict(c)[valueId]$ is the value, from the column domain, encoded as $valueId$, while $ValId(c)[rowId]$ is the $valueId$ of the column at position $rowId$. These two arrays are sufficient to provide transactional durability: we refer to these as *primary* data. SOFORT uses SCM as memory and storage at the same time by keeping primary data, accessing it, and updating it in-place in SCM. In other words, the working copy and the durable copy of the data are merged. Other data structures are required to achieve reasonable performance including, for each column c, a dictionary index $iDict(c)$ that maps values to value ids, and for each table, a set of multi-column inverted indexes $iValId(S_c)$ that map sets of value ids to the set of corresponding row ids. We refer to these structures as *secondary* data since they can be reconstructed from the primary data. SOFORT can keep secondary data in DRAM, in SCM, or in a hybrid SCM-DRAM format. In this work however, we assume that all secondary data structures are placed in DRAM for optimal query performance.

2.5 Recovery Techniques

In-memory DBMSs recover by rebuilding DRAM-based data structures from a consistent state persisted on durable media. The persisted state consists of a copy (checkpoint) of the database state at a particular point in time and a log of subsequent updates. Recovery consists of reloading portions of the persisted state, applying subsequent updates, and undoing the effects of unfinished transactions. The major bottleneck in this approach is reloading the persisted state from disks to main memory, which may take a significant amount of time for large database instances. SOFORT removes this bottleneck by keeping primary data in SCM and directly operating on it.

SOFORT recovers by first recovering primary data that is persisted in SCM at a negligible cost, then undoing the effects of unfinished transactions [29]. The last phase of the recovery procedure, that is, rebuilding secondary data, can be handled in two different ways: in the first approach, denoted *Synchronous Recovery*, SOFORT does not accept requests until the end of the recovery process, i.e., until all secondary data structures have been rebuilt. The main advantage of this approach is that it rebuilds secondary data structures as fast as possible since all system resources are allocated to recovery. However, this approach suffers from the fact that the database is not responsive during the whole recovery period – which might be long as its duration depends directly on the size of secondary data structures to be rebuilt.

To achieve instant recovery, SOFORT adopts a *crawl before run* approach: it uses primary data, which is recovered at a negligible cost, to answer queries, while secondary data structures, whose purpose is to speed up query processing, are rebuilt in the background. For example, lookups to a dictionary index and an inverted index are replaced by scans of the corresponding dictionary array and the value ids arrays, respectively. Partially rebuilt DRAM-based indexes can be progressively leveraged as they are being rebuilt. For instance, a regular dictionary index lookup is replaced by the following sequence: look up the partially rebuilt dictionary index; if the value is not found, then scan only the portion of the dictionary array that has not been indexed yet. Hybrid SCM-DRAM indexes, however, need to be fully recovered in order to be leveraged. The main advantage of this approach is that it enables instant recovery, i.e., instant responsiveness of the database after failure. However, it takes the database a considerable amount of time, longer than for synchronous recovery, to approach peak performance observed before failure. This is because system resources are split between recovery and query processing, while in synchronous recovery, all system resources are allocated to recovery.

Discussion

From the above description of different recovery techniques, we observe that the bottleneck of recovery is shifted in SCM-enabled databases from reloading primary data from durable media to main memory, to rebuilding secondary data structures. Optimizing the latter did not get much attention in the past because it was shadowed by the former more prominent bottleneck. In this paper we propose

a solution to overcome this new bottleneck. Although synchronous recovery and instant recovery exhibit interesting advantages, they both suffer from (different) noticeable shortcomings. We try to cure the shortcomings of both approaches by achieving instant responsiveness and reaching peak performance quickly, well before the end of the recovery process. The following section explains in details how we achieve this.

3. ADAPTIVE RECOVERY

Adaptive recovery is based on the observation that not all secondary data structures are equally important to a specific workload. Therefore, rebuilding the most relevant ones first should significantly improve the performance of the workload run concurrently with the recovery process, eventually improving the overall recovery performance. The following questions arise: What are the characteristics of secondary data structures? How are secondary data structures rebuilt? How can the *benefit* of a secondary data structure be measured and estimated for a specific workload?

In this section, we answer these questions in order. We discuss the recovery procedure of secondary data structures based on a characterization of their recovery properties. Then, we introduce benefit functions centered on the usefulness and rebuild cost of these structures given a recovery workload. As the recovery process is usually a very short period of time compared with the time the system is in normal operation, the benefit functions must quickly adapt to the current workload to maximize performance of the system during recovery. A recovery manager will take into account the workload right before the crash and the workload during recovery to decide the recovery procedure of the secondary data structures by answering these questions: what resources are allocated to the recovery process? does the recovery happen in a background process or synchronously with the statements needing them? and what is the recovery order?

3.1 Recovery of Secondary Data Structures

Secondary data structures can reside in SCM or in DRAM. The decision for this characteristic can be decided as part of the database physical design process, based on cost estimated that takes into account the schema, the base data properties, the workload, and how fast the recovery process must be. In a robust system, when recovery must be guaranteed to be almost instantaneous, it may be desired to have all primary data and secondary data structures in SCM for fast recovery. However, this approach will significantly impact query performance as discussed in Section 2.3.

Secondary data structures, such as indexes, materialized views, intermediate cached results are structures that can be completely rebuilt from primary data. In this work, we identify and assume the following general properties of using and maintaining secondary data structures:

1. query processing can be correctly done (i.e. correct result sets) without using any secondary data structures;
2. secondary data structures are used for improving global performance;
3. the decisions to build and use them is cost-based taking into account the cost of maintaining them, and the performance benefit their usage will bring;
4. for a particular statement, the decision to use an available secondary data structure is cost-based;
5. secondary data structures may be useful when they are only partially built, and the query optimizer will build correct plans capable of compensating for missing data. For example, this can be implemented using partially materialized views or cracked indexes.

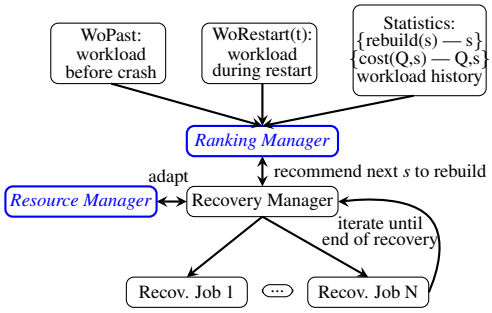


Figure 6: Recovery framework of secondary database objects.

In an SCM-DRAM hybrid system such as SOFORT, secondary data structures can be built and stored as follows:

- *Transient data structures* fully residing in DRAM. These structures must be fully rebuilt from primary data at restart to reach a consistent state similar to the state before the crash.
- *Persistent data structures* fully residing in SCM and persisted in real-time. These structures are available to be used by query processing, almost instantaneously, after a *recover* process which includes sanity checks and repairs if needed.
- *Hybrid transient-persistent data structures* partially residing in DRAM, and partially in SCM (e.g., the FPTree [30]). The persistent part must go through a *recovery* process which includes sanity checks and repairs if needed. The transient part is rebuilt based on one or both of the persistent part and the primary data.

Regardless of the type of a secondary data structure, the rebuild or reload can be executed to completion for the whole structure, or it can be partially done on a need-to-do basis based on the workload, e.g., data cracking techniques [19]. Each secondary data structure can be rebuilt right after restart or on demand, e.g., when this structure is first needed by a transaction (for writes or reads). These decisions should be cost-based, balancing the performance benefit for the workload during recovery with the overhead of recovery, i.e., reloading and rebuilding.

3.2 Benefit Functions for Secondary Data

Regardless of where a secondary data structure resides or what the recovery process for a specific secondary data structure is, we can compare and rank them based on the usefulness to past and current workloads, and how expensive it is to rebuild them at recovery time. Our goal is to maximize performance of the workload run in parallel with the recovery process. In real database systems, the workload right after restart can be similar to the workload at crash time (e.g., unfinished transactions are resubmitted after restart), it can be very different (e.g., the system must execute a specific restart workload), or a mixed workload including both special restart transactions and resubmitted transactions. As the recovery time can be very short, the benefit functions for secondary data structures must quickly adapt to the current workload and available resources for recovery process.

At first glance, using benefit functions to rank the usefulness of secondary data structures is similar to techniques used for indexes and materialized views selection. Index selection has been studied in the research literature, and most commercial database systems have a physical design adviser tool (see, for example, [36] for a survey). The main differences between index selection and our ranking problem during recovery are:

1. index selection works with a large, known workload that is representative of normal usage of the database system;
2. algorithms for index selection are run off-line, and can take hours;

3. index selection deals with multi-objective functions, for maximizing workload performance and minimizing the total space requirements for selected indexes;
4. the interaction among recommended indexes is implicitly captured in the algorithms employed by index advisers which heuristically enumerate subsets of indexes and compare them based on their estimated benefit. In [36], authors address the issue of measuring interactions among indexes for a given set of recommended indexes, with experiments showing that this particular method can take minutes to run for a medium size set of indexes;

In contrast, computing the benefit of secondary data structures during recovery is characterized by the following:

1. it is applied to a dynamic restart workload that can be very different from the observed normal workload; this workload is run for a very short period of time;
2. it is applied to a well defined schema of secondary data structures, all of which must be rebuilt before the end of the recovery process. However, many of them are not useful to this restart workload.

Our proposed method is based on computing when the recovery starts, at time 0, an original ranking based on the immediate past workload, and dynamically adjusting this ranking, during recovery, based on the current workload. This dynamic ranking function can use a benefit function that captures the estimated usefulness of an index compared to other indexes. Below, we discuss the general algorithm for ranking based on a given benefit function. We then introduce two possible benefit functions that can be used for ranking. We compare their properties and discuss how these can be computed, on-line, during the recovery process.

We denote by $\mathcal{S}_{SCM}$ the set of secondary data structures residing in SCM that do not need to be rebuilt as they are available almost instantly at restart time, and $\mathcal{S}$ the set of all secondary data structures defined in the database schema. During recovery, all secondary data structures in $\mathcal{S} \setminus \mathcal{S}_{SCM}$ must be rebuilt.

We assume that a *query benefit* function is available, $\textit{Benefit}(s, Q, \mathcal{S})$ which estimates the benefit of a secondary data structure $s \in \mathcal{S}$ with respect to the set $\mathcal{S}$ and the statement Q, such as the ones we will introduce below, namely $\textit{Benefit}_{indep}$ and $\textit{Benefit}_{dep}$. Given a workload W, a multi-set of statements (i.e., repeated statements are considered) and $\textit{Benefit}(s, Q, \mathcal{S})$, we define the *benefit* of a secondary data structure s with respect to $\mathcal{S}$ for W as:

$$\textit{Benefit}(s, W, \mathcal{S}) = \Sigma_{Q \in W} \textit{Benefit}(s, Q, \mathcal{S}) \tag{1}$$

The rank of a secondary data structure s at restart time $\textit{rank}(s, 0)$ can be computed based on the observed workload in the immediate past of the crash *WoPast* and taking into account the estimated rebuild cost (denoted here by $\textit{rebuild}(s)$):

$$\begin{aligned} \textit{rank}(s, 0) = &\textit{Benefit}(s, \textit{WoPast}, \mathcal{S}) \\ &- \textit{rebuild}(s) \end{aligned} \tag{2}$$

During recovery, the benefit of an index changes as the current workload, run in parallel with the recovery procedure, progresses. The content of the workload since the restart and up to current time t, $\textit{WoRestart}(t)$, can be used to adjust the ranking $\textit{rank}(s, t)$ of the yet-to-be-built structure $s \in \mathcal{S}$. The ranking function must be effective for all scenarios: the workload after restart is the same as before, very different, or a combination. We propose a weighted average with adjusted weights based on the number of statements observed since the restart. As the recovery manager considers what to recover next, rankings are adjusted and next best ranked index is rebuilt. Because rebuild cost is considered in the $rank()$ formula, the ranks can be negative for data structures for which the rebuild cost exceeds the benefit to the workload. With $n = \textit{sizeof}(\textit{WoRestart}(t))$, the following formula computes the ranking at time t:

$$\begin{aligned} \textit{rank}(s, t) = &\alpha(n) * \textit{Benefit}(s, \textit{WoPast}, \mathcal{S}) \\ &+ (1 - \alpha(n)) * \textit{Benefit}(s, \textit{WoRestart}(t), \mathcal{S}) \\ &- \textit{rebuild}(s) \end{aligned} \tag{3}$$

$\alpha(n)$ can be defined such that it decays exponentially with n (e.g., $\alpha(n) = \alpha^n$ with $0 \leq \alpha \leq 1$) to increase the weight of the current workload as more statements are seen during the recovery period.

3.3 Benefit Functions $\textit{Benefit}_{dep}$ and $\textit{Benefit}_{indep}$

The above formulas require that a query benefit function, denoted $\textit{Benefit}(s, Q, \mathcal{S})$, is available. We use in this work two benefit functions, one, $\textit{Benefit}_{indep}$, based solely on the independent effect of an index s on a statement Q, while the other, $\textit{Benefit}_{dep}$ captures the effect of the index s with respect to the whole set of defined secondary data structures $\mathcal{S}$.

We assume that the query optimizer can generate an optimal plan, denoted by $plan(Q, I)$ and its estimated cost, denoted by $\textit{Cost}(Q, I)$, for a statement Q when only a subset of indexes $I \subseteq \mathcal{S}$ are available for the query execution. If a secondary data structure $s \in I$ is used in $plan(Q, I)$, we use the notation $s \in \mathcal{S}(plan(Q, I))$. We assume that there exists a well-behaved optimizer which consistently builds the same optimal plan in the presence of the same indexes: i.e., if $I_1 \subseteq I_2$, and $\mathcal{S}(plan(Q, I_2)) \subseteq I_1 \subseteq I_2$, then $plan(Q, I_2) = plan(Q, I_1)$.

We denote by t_Q, the time during $WoRestart(t)$ when the query Q was run, hence $t_Q \leq t$; and by $S(t)$ the set of indexes available for query execution at time t. Note that $S(0) = S_{SCM}$, as the only secondary data structures available at the start of recovery are the indexes which are instantly recovered (i.e., stored in SCM).

$$\begin{aligned} &\text{for } Q \in \textit{WoRestart}(t), WoPast: \\ &\textit{Benefit}_{indep}(s, Q, \mathcal{S}) = \textit{Cost}(Q, \mathcal{S}_{SCM}) \\ &\qquad\qquad - \textit{Cost}(Q, \mathcal{S}_{SCM} \cup \{s\}) \end{aligned} \tag{4}$$

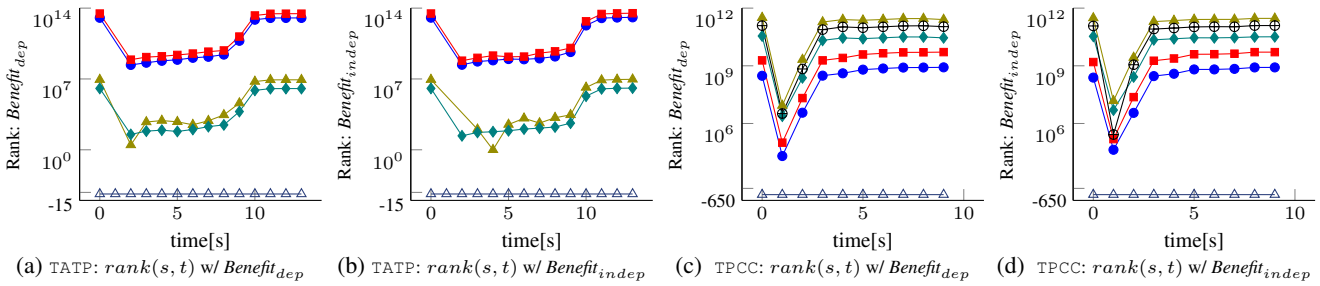
$\textit{Benefit}_{indep}$, in Eq. 4, is defined completely in isolation from and independent of the other indexes in $S \setminus S_{SCM}$. Using this function, $rank(s, t)$ (cf. Eq. 3), captures what is the benefit of having available just $\mathcal{S}_{SCM} \cup \{s\}$ when the workloads $WoPast$ or $WoRestart(t)$ are run.

$$\begin{aligned} &\text{for } Q \in \textit{WoRestart}(t): \\ &\textit{Benefit}_{dep}(s, Q, \mathcal{S}) = \textit{Cost}(Q, \mathcal{S}(t_Q)) \\ &\qquad\qquad - \textit{Cost}(Q, \mathcal{S}(t_Q) \cup \{s\}) \end{aligned} \tag{5}$$

$$\begin{aligned} &\text{for } Q \in \textit{WoPast}: \\ &\textit{Benefit}_{dep}(s, Q, \mathcal{S}) = \\ &\begin{cases} \textit{Cost}(Q, \mathcal{S}_{SCM}) - \textit{Cost}(Q, \mathcal{S}), & s \in \mathcal{S}(plan(s, Q)) \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{6}$$

In contrast, $\textit{Benefit}_{dep}$, in Eq. 5, captures the dependency of indexes that are used together in the best plans: for any index s participating in the optimal plan of Q at the time t_Q, i.e., $s \in \mathcal{S}(plan(Q, \mathcal{S}(t_Q) \cup \{s\}))$, the cost difference between the current plan and the plan using s, i.e., $\textit{Cost}(Q, \mathcal{S}(t_Q)) - \textit{Cost}(Q, \mathcal{S}(t_Q) \cup \{s\})$, is added to $rank(s, t)$ as per in Eq. 3.

To show how ranking is progressing during recovery, we picked a set of interesting indexes from the TATP and TPCC benchmarks used in the evaluation section and plotted the evolution of their $rank(s, t)$ functions during recovery in Figure 7, for both benefit functions $\textit{Benefit}_{dep}$ and $\textit{Benefit}_{indep}$. We choose $\alpha(n) = 0.99^n$,



(a) TATP: $rank(s,t)$ w/ $Benefit_{dep}$ (b) TATP: $rank(s,t)$ w/ $Benefit_{indep}$ (c) TPCC: $rank(s,t)$ w/ $Benefit_{dep}$ (d) TPCC: $rank(s,t)$ w/ $Benefit_{indep}$

Figure 7: *rank(s,t)* for TATP (20 Mio subscribers) and TPCC (50 Warehouses) during recovery : $Benefit_{dep}$ vs. $Benefit_{indep}$.

which decays exponentially as the number of observed statements n increases. $\mathcal{S}$ has 51 dictionary indexes (one per column) and 5 inverted indexes in TATP, and 94 dictionary indexes and 10 inverted indexes in TPCC, all of which are stored in DRAM. In these figures, each $rank(s,t)$ function has the same line style for the same index s. Note that indexes that have the same rank using $Benefit_{indep}$ (Figures 7b & 7d) may not have the same rank using $Benefit_{dep}$ (Figures 7a & 7c). However, for these particular benchmarks, both query benefit functions result in the same ranking among indexes, which gives the same rebuild order.

If s is not used in any optimal plans, its benefit for the workload is 0 (for both $Benefit_{indep}$ and $Benefit_{dep}$), hence $rank(s,0)$ is negative because of the $rebuild(s)$ cost, and $rank(s,t)$ $(= rank(s,0))$ is also negative until s is rebuilt during recovery. Figure 7 shows a set of unused indexes during $WoPast$ and $WoRestart(t)$ and their negative ranking.

As we assume that the decision to use a secondary data structure during query processing is cost-based, the estimated costs $Cost(Q, \mathcal{S}_{SCM})$, $Cost(Q, \mathcal{S}_{SCM} \cup \{s\})$, and $Cost(Q, \mathcal{S})$ needed in $Benefit_{dep}$ and $Benefit_{indep}$ functions are computed by the query optimizer during the query optimization of the statement Q. For example, in Eq. 5, $Cost(Q, S(t_Q))$ is exactly the estimated cost of the best plan executed when Q was run at time t_Q during recovery, with the available indexes in $S(t_Q)$. The workload statistics capturing $WoPast$ can be collected as statement identifiers, with frequencies, for a fixed window of time. Such statistics are usually recorded anyway by real systems, for example, for security purposes, and performance tuning.

3.4 Recovery Manager

In SOFORT, recovery starts with a *recovery* process of all persistent data structures residing in SCM. This is an almost instantaneous process. As all primary data is persisted in SCM, SOFORT will not be available for new transactions until this initial recover process is finished. According to definitions in SLAs, a system is considered as *available* if users are able to connect. Hence, in SOFORT, availability is almost instantaneous after software failures. For *Instant Recovery* and *Adaptive Recovery*, SOFORT allows new connections right after recovering primary data. These new connections yield, at each time t, the restart workload denoted by *WoRestart(t)*. In parallel, SOFORT performs the recovery of secondary data structures which need to be rebuilt, i.e., the ones residing in DRAM. As resource allocation between the usual restart workload and the recovery process adapts to the current state, resources can be fully allocated to the recovery process. All secondary data structures need to be eventually rebuilt even if they are not useful to the current workload. Under these constraints, the main goal is to maximize throughput during recovery time.

The recovery process, as implemented in SOFORT, for secondary data structures can be achieved in different ways: (1) recover all secondary data structure before allowing any connections (referred here as *Synchronous Recovery*); (2) recover a secondary data structure right before executing a statement that needs it. This decision is cost-based on the rebuild time and the benefit to the statement; (3) in a background recovery process that is invoked by the recovery manager with either fixed or adjustable resources, using or not using a ranking algorithm (referred here as *Adaptive Recovery*). In the most optimized operation mode, the recovery manager consults a ranking component that provides an updated ranking of the yet-to-be built secondary data structures, and adapts the resource allocation between query processing and recovery based on the new ranking.

Algorithm 1 *RecoveryProcedure*

1: R = maximum available resources
2: **while** $\exists$ *secondary data structure* yet-to-be built and $\exists$ free resources out of R **do**
3: $I = NextToRebuild(R)$
4: /* Adjust resources based on I */
5: R' = available resources to recover I
6: rebuild a subset of I using R'

Algorithm 2 *NextToRebuild*(R)

1: INPUT: R available resources
2: OUTPUT: a set I of *secondary data structure* to recover next
3: t = currrent time
4: $n = sizeof(WoRestart(t))$
5: **for** $\forall s$, a *secondary data structure* not yet rebuilt **do**
6: $rank(s,t) = \alpha(n) * Benefit(s, WoPast, \mathcal{S})$
7: $+(1-\alpha(n)) * Benefit(s, WoRestart(t), \mathcal{S}) - rebuild(s)$
8: I = top ranked secondary data structures to be built using R resources

Figure 6 shows the general recovery framework which handles the recovery of secondary data structures. Right after restart, the recovery manager starts recovery procedures for the highly ranked secondary data structure. The recovery process is run in parallel with other statements unless all resources are allocated to the recovery process. The ranking at restart time $rank(s,0)$ is based on the workload before the crash. the recovery manager can use adaptive or static resource allocation as described in Algorithm 1 (line 4). As recovery jobs finish and resources become available to the recovery process again (line 5 in Algorithm 1), a current ranking is computed and next secondary data structures candidates are chosen using

Algorithm 2. Secondary data structures with negative benefits will be built last. Note that one way to adapt resource allocation is based on the observed benefit of the most highly ranked secondary data structure: the lower this benefit, the more resources we release to query processing.

4. EVALUATION

Experimental Details

In our experiments, we use the SCM emulator that is described in Section 2.2. We fix the latency of SCM to 200 ns which is more than 2 x higher than the latency of DRAM (90 ns) in the emulation system. Except if mentioned otherwise, we bind all tests to a single socket (with 8 cores) to isolate NUMA effects from the effects of SCM's higher latency. Hyperthreading is disabled.

We use the Telecom Application Transaction Processing (TATP) benchmark [4] and the TPCC benchmark. TPCC is the industry standard OLTP benchmark. Its schema encompasses nine tables and it consists of five transaction templates. TATP is a simple but realistic OLTP benchmark that simulates a telecommunication application. The schema of the benchmark comprises four tables and the benchmark consists of seven transaction templates. Unless specified otherwise, we run the following query mixes in the experiments below: (1) the ORDER STATUS (50%) and STOCK LEVEL (50%) queries of TPCC, and (2) the GET SUBSCRIBER DATA (80%) and UPDATE LOCATION (20%) of TATP. The experiments were run using TPCC scale factor 32 (i.e, 32 Warehouses) and TATP scale factor 500 (i.e., 5 Mio Subscribers), which corresponds to an initial database size of 10 GB and 6 GB, respectively. We run the benchmarks with eight users (clients) in all experiments. Table 2 shows the labels used for each experimental configuration, and helps understand the legends of the figures in this section.

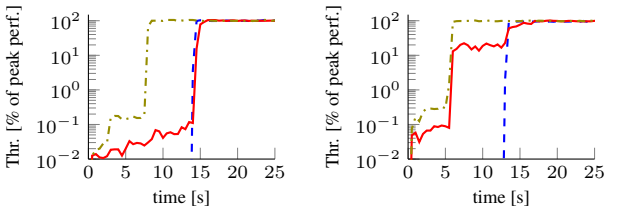
Table 2: Experimental Configurations.

Experiment	*Query Processing Resources*	*Recovery Resources*	*Ranking*
$rk.Qx.Rz$	static x cores	static z cores	yes
$\neg rk.Qx.Rz$	static x cores	static z cores	no
$rk.Q^{ad}.Rz^{ad}$	adaptive	adaptive, start with z cores	yes

Ranking Algorithm

In the first experiment, we run the aforementioned benchmarks and simulate a crash. Then, we observe the recovery behavior of different recovery strategies, where the restart workload *WoRestart* is the same as the workload observed before the crash. The results are depicted in Figure 8.

- $\neg rk.Q0.R8$: the recovery process is run synchronously. All resources are allocated to the recovery process until its end, and new transactions are not allowed before the end of recovery. We consider this strategy as the baseline.
- $\neg rk.Q6.R2$: the recovery process is run in parallel with the recovery workload *WoRestart*, and the secondary data structures are not ranked, i.e., they are recovered by prioritizing referenced indexes. Although a noticeable increase in performance occurs before the end of recovery, this strategy performs worse than the synchronous one for TATP. Indeed, 15 s after the start of recovery, only 0.4 million transactions were executed, which is less than the 0.9 million transactions executed in $\neg rk.Q0.R8$. This result emphasizes the importance of ordering the rebuild of secondary data structures according to their importance for the workload. A careless ranking strategy can backfire and prove to be harmful to recovery performance.
- $rk.Q^{ad}.R6^{ad}$: the recovery process is run in parallel with the recovery workload *WoRestart*, and the secondary data structures are rebuilt based on our ranking function. Adaptive resource allocation is enabled. This strategy outperforms both previous strategies as throughput gradually increases to reach peak performance. The performance of query processing surges earlier than in $\neg rk.Q6.R2$. 15 s after the start recovery, 6.4 million and 0.83 million transactions were executed for TATP and TPCC respectively, i.e., 5.5 million and 0.67 million transactions more than in $\neg rk.Q0.R8$, respectively.



(a) TATP (b) TPCC

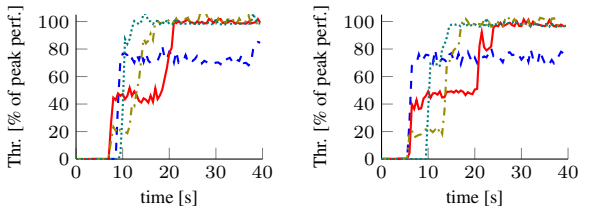
Legend Entry	TATP		TPCC	
	Rec. End	#TXs t=15 s	Rec. End	#TXs t=15 s
$\neg rk.Q0.R8$	13.9 s	0.9 M	13 s	169 K
$\neg rk.Q2.R6$	15.5 s	0.4 M	16.5 s	246 K
$rk.Q^{ad}.R6^{ad}$	63.7 s	6.4 M	65.2 s	838 K

Figure 8: Different recovery strategies with logarithmic scale.

Another important dimension is resource sharing between recovery and query processing. We experiment with several static resource allocation configurations and report the results in Figure 9. We notice that the more resources we allocate to the recovery process the better the performance (cf. legend table): at time 30 s, the best configuration ($rk.Q0.R8$) executes 1.29× more transactions than the worst configuration ($rk.Q6.R2$). This result emphasizes the importance of resource allocation. We also observe that the figures are shaped like stairs, where a peak corresponds to the end of the recovery of an important secondary data structures group. The reason why the increase in throughput is in stair-steps and not linear is the design of the benchmark: all users run the same mix of queries, which means that even if only one single query did not regain its optimal performance, it will drain the query throughput down across all users. Hence, substantial improvements in throughput occur only once the performance of all queries improves. This setup is not in favor of our approach, as earlier benefits of our approach could be observed in a different setup where users run different queries from each other, or run the same queries on different copies of the data.

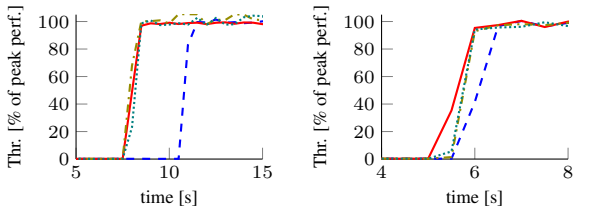
Although query throughput is less than 1% of peak performance at the beginning, this still represents thousands of transactions per second, which allows to run high priority queries. We also notice that the more resources allocated to recovery, the sooner the first stair-step appears and the lower it is. This is due to the fact that more resources for recovery means faster rebuild of secondary data structures (hence faster appearance of stair-steps) and slower query processing due to lack of resources (hence, the lower the stair-steps). The last peak that brings each configuration back to peak performance takes place at the end of the recovery process, which again happens sooner with more recovery resources, but configurations

[4] http://tatpbenchmark.sourceforge.net/



Legend Entry	TATP		TPCC	
	Rec. End	#TXs t=30 s	Rec. End	#TXs t=30 s
$rk.Q6.R2$	40 s	13.3 M	42.6 s	1.27 M
$rk.Q4.R4$	20.4 s	13.8 M	23.6 s	0.96 M
$rk.Q2.R6$	16.4 s	15.5 M	16.6 s	1.1 M
$rk.Q0.R8$	11.6 s	17.1 M	13.7 s	1.25 M

Figure 9: Static resource allocation impact on recovery.



Legend Entry	TATP		TPCC	
	Rec. End	#TXs t=15 s	Rec. End	#TXs t=8 s
$rk.Q^{ad}.R2^{ad}$	64.3 s	3.8 M	64.8 s	196 K
$rk.Q^{ad}.R4^{ad}$	63.5 s	6.1 M	64.3 s	273 K
$rk.Q^{ad}.R6^{ad}$	63.6 s	6.4 M	65.2 s	220 K
$rk.Q^{ad}.R8^{ad}$	63.2 s	6.1 M	64.9 s	220 K

Figure 10: Adaptive resource allocation impact on recovery.

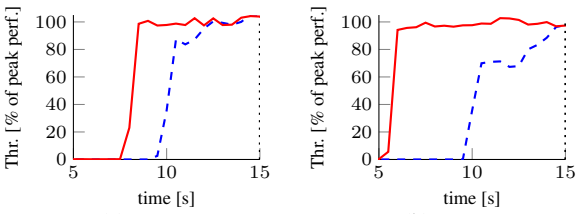
with less recovery resources compensate with higher stair-steps. An intuition that stems from this result is that a combination of all configurations, such that recovery starts with more resources and releases them gradually whenever query performance surges might lead to a better recovery performance.

Adaptive Resource Allocation Algorithm

In this experiment, we investigate the efficiency of our adaptive resource allocation algorithm. The recovery manager is given specific resources and is free to use them fully and/or return them partially/-fully to query processing. Basically, the recovery manager decides to give back recovery resources to query processing once it detects, based on the benefit value of the most highly ranked index, that enough secondary data structures have been rebuilt to allow the system to run at nearly full speed. The remaining secondary data structures are recovered in the background whenever the CPU idles.

Figure 10 summarizes the results of this experiment. Theoretically allocating more resources to recovery should enable the database to reach peak performance sooner. Notably, we observe that all other recovery configurations perform nearly equally, except $rk.Q^{ad}.R2^{ad}$ which performs worse. This is explained by two factors. First, the set of important indexes is small in our benchmarks, which means that only a few recovery resources are needed. Second, having more resources allocated to query processing allows to gain more knowledge of the current workload, hence swiftly narrowing down the set of important indexes to recover. The recovery manager can then invest all available resources in rebuilding as quickly as possible these indexes, and then give back all the resources to query processing and continue rebuilding the remaining secondary data structures in the background. As a consequence, the system reaches peak performance before the end of the recovery process. An interesting observation is that the stair-steps in the figure are not as explicit as in static resource allocation. Actually, the gradual increase in throughput results from the fact that some resources finish their job and are released earlier than others.

Figure 11 shows a comparison between the best configuration for static resource allocation and the best configuration for adaptive resource allocation. We observe that at time 15 s, $rk.Q^{ad}.R8^{ad}$ executes $1.41\times$ and $2.26\times$ more transactions than $rk.Q0.R8$, for TATP and TPCC respectively. Also, for static resource allocation, peak performance is regained only at the end of the recovery process while for adaptive resource allocation, peak performance is regained before the end of the recovery process. From this point on, unless specified otherwise, we experiment with adaptive resource



Legend Entry	TATP		TPCC	
	Rec. End	#TXs t=15 s	Rec. End	#TXs t=15 s
$rk.Q0.R8$	11.6 s	4.3 M	13.7 s	373 K
$rk.Q^{ad}.R8^{ad}$	63.2 s	6.1 M	64.9 s	843 K

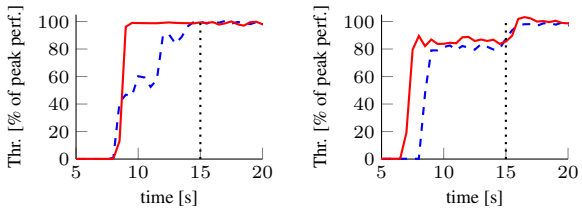
Figure 11: Best of static vs. best of adaptive resource allocation.

allocation enabled. The adaptive approach brings a noticeable enhancement in recovery performance over the static one, but has the drawback of prohibiting query processing at the beginning of recovery in configuration $rk.Q^{ad}.R8^{ad}$. For scenarios where allowing query processing right from the beginning of recovery is crucial, configurations $rk.Q^{ad}.R6^{ad}$ and $rk.Q^{ad}.R4^{ad}$ might be more interesting. Moreover, allowing query processing as early as possible has the advantage of enriching knowledge about $WoRestart$, hence allowing to adapt to workload changes.

Resilience to Workload Change

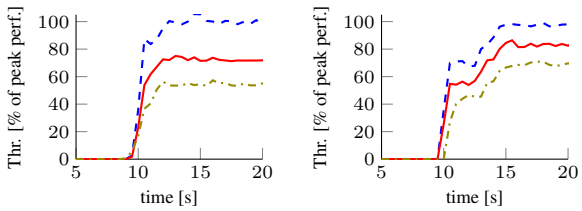
This experiment is dedicated to studying the effect of changing workloads during recovery. To do so, we execute different workloads before and after the crash: before failure, the full original TATP and TPCC mixes are run; after failure however, only the TATP and TPCC queries used in the previous experiments are run. This approach has the benefit that the workload before failure uses a superset of the indexes used by the workload after failure. Indeed, a subset of the indexes is used more often while others are not used anymore after failure, which makes it a good example to study the importance of considering *WoPast* and *WoRestart*. We run configuration $rk.Q^{ad}.R6^{ad}$ to allow adaptive recovery to gain knowledge of *WoRestart*.

Figure 12 illustrates the results of this experiment. We observe that at time 15 s, the adapting approach executes $1.27\times$ and $1.25\times$



Legend Entry	TATP		TPCC	
	Rec. End	#TXs t=15 s	Rec. End	#TXs t=15 s
no adapt to WoRes.	28.1 s	4.4 M	59.5 s	498 K
adapt to WoRes.	49.4 s	5.6 M	58.3 s	624 K

Figure 12: Impact of workload change during recovery.



Legend Entry	TATP Rec. End	TPCC Rec. End
200 ns	11.6 s	13.8 s
450 ns	11.2 s	14.1 s
700 ns	11.2 s	14.1 s

Figure 13: SCM's latency impact on recovery performance.

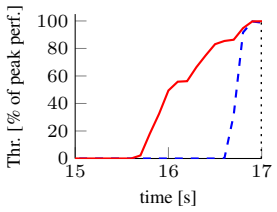
more transactions than the non adapting one, for TATP and TPCC respectively. This results clearly show the benefits of adapting to changing workloads during recovery. Also, not taking into account *WoRestart* in the ranking function could lead to a scenario similar to the instant recovery approach, which might turn out to be harmful to recovery performance. Other scenarios, such as using completely different secondary data structures before and after failure, would highlight much more the importance of considering *WoRestart* in the benefit function.

Impact of SCM *Latency*

In this experiment we run the TATP and TPCC mixes described in Section 4 with configuration $rk.Q0.R8$, and vary the latency of SCM between 200 ns and 700 ns. We report the results in Figure 13. We notice that while the peak performance of the system suffers from higher SCM latencies, recovery does not. Indeed, primary data is accessed sequentially in SCM to rebuild secondary data in DRAM. The system reaches its peak performance approximately at the same time for all latency configurations. In brief, the benefits of our approach are almost independent of the latency of the SCM. Note that the performance degradation is due to the materialization of rows, where at least one random SCM access per column is executed.

Limits of Adaptive Recovery

To conduct a worst case analysis, we design a synthetic benchmark that encompasses 10 tables with 10 integer columns each. All tables have one million rows of distinct values. Similar to the previous benchmarks, eight users run a mix of queries. The mix consists of 100 queries, each of which executes a simple select with a predicate



Legend Entry	Rec. End	#TXs t=17 s
$\neg rk.Q0.R8$	16.6 s	0.29 M
$rk.Q^{ad}.R8^{ad}$	16.6 s	0.84 M

Figure 14: Worst case analysis: Adaptive recovery Vs. Synchronous recovery.

on a single, distinct column. Overall, all columns are uniformly queried. We report the experimental results in Figure 14. Theoretically $rk.Q^{ad}.R8^{ad}$ and $\neg rk.Q0.R8$ should perform nearly equally. This is because all indexes have similar benefits and are equally important to the workload. Hence, all orders of recovery should yield similar recovery performance. Surprisingly, the difference in performance is fairly large: $rk.Q^{ad}.R8^{ad}$ executed 2.9× more transactions than $\neg rk.Q0.R8$ by the end of recovery (17 s). This is thanks to adaptive resource allocation, that is, resources are released as soon as the recovery job queue is empty, while in synchronous recovery no resources are released until all recovery jobs are finished. We conclude that the performance of synchronous recovery is a lower bound for the performance of adaptive recovery.

5. RELATED WORK

We divide related work into two categories: efforts to leverage SCM to improve databases recovery performance, and traditional main-memory database recovery related works.

5.1 SCM and Recovery

In our previous work [29], we propose a radical change in the database architecture by leveraging SCM as memory and storage at the same time, thanks to which we achieve fast data recovery, as data can be directly accessed in SCM and does not need to be fetched from disk to DRAM. We built a prototype called SOFORT, that we extend in this paper. However, SOFORT's recovery in [29] is limited as it relies on rebuilding all secondary data structures synchronously before allowing any new connections. This can result in high recovery times for schemas with many and large secondary data structures. Thereafter, we proposed in [31] an approach to remedy this issue, which is to trade between query performance and recovery performance by persisting some or all of the secondary data structures in SCM. While this approach is interesting as it enables near-instant recovery directly at maximum throughput, it compromises query performance as the overhead of persisting secondary data structures is not negligible: 14% and 30% when persisting 40% and 100% of secondary data structures, respectively. The gain during recovery is dwarfed by the loss in query processing performance over time. In this paper, we alleviate both limitations of [29, 31] by using an adaptive recovery approach that relies on a ranking function of secondary data structures and an adaptive resource allocation algorithm, allowing for significantly better recovery performance without compromising query performance.

Schwalb et al. [37] proposed Hyrise-NV, an SCM-enabled version of the Hyrise main-memory storage engine [16]. Hyrise-NV persists all its data structures in SCM, including secondary indexes, while SOFORT adopts an SCM-DRAM hybrid architecture. Kimura [23]

proposed FOEDUS, a novel OLTP-oriented database system designed to take advantage of upcoming servers with thousands of cores and very large amounts of SCM. Contrary to SOFORT, FOEDUS uses DRAM as a cache for SCM and provides durability through traditional write-ahead logging. Arulraj et al. [1] investigated three ways of leveraging SCM in databases, namely in-place updates, copy-on-write, and log-structured updates. They conclude that in-place updates outperform the two other approaches. In addition, they study the corresponding recovery techniques and demonstrate that, like in SOFORT, there is no need to reload primary data nor to apply a redo log since transaction changes are made persistent at commit time. However, they rely on traditional logging techniques. To remedy this issue, they proposed Write-Behind Logging [2], a minimalistic logging technique that, coupled with multi-versioning, persists new tuples before writing the corresponding log entry. This enables it to stores a reference instead of the full tuple in the log. However, the authors do not take into consideration the rebuilding of DRAM-based data structures.

Chatzistergiou et al. [4] propose REWIND, a log-based user-mode library that manages persistent data structures in SCM in a recoverable state. They show that REWIND-based recovery outperforms traditional I/O based recovery techniques by up to two orders of magnitude. However, REWIND incurs an overhead to the overall system performance and handles only SCM-based data. Pelley et al. [34] propose two architecture alternatives, namely in-place updates and NVRAM group commit, to optimize OLTP durability management and speed-up recovery. In the in-place updates architecture, writes are done directly to SCM, while reads are optionally performed through a software-managed DRAM cache. In the NVRAM group commit architecture, writes are buffered in DRAM and propagated to SCM when a transaction batch commits. Both architectures do not need a redo phase during recovery, but they still rely on traditional write-ahead logging for the undo phase. In contrast, SOFORT treats SCM as in the same level of DRAM, and goes further into getting rid of a traditional logging.

Several works focused on improving the logging infrastructure of traditional database systems [11, 12, 41, 18]. Our approach is radically different: We propose a green field approach that leverages SCM as memory and storage at the same time for the whole database, which enables to do away with a transactional log. Finally, several works investigated SCM-optimized persistent data structures [39, 30, 6, 45], while other works investigated optimizing database algorithms, such as sorting and joins, for SCM [5, 40]. These works are orthogonal, though related to ours. Indeed, to leverage the full potential of SCM, there is a need to redesign classical main-memory data structures to make them persistent in SCM, and to optimize database algorithms to account for the read/write latency and bandwidth asymmetry of SCM.

5.2 Recovery of main-memory databases

Since our design avoids traditional logging, most existing recovery techniques do not apply to our work. Nevertheless, for the sake of completeness, we briefly review state of the art database recovery techniques.

Garcia-Molina et al. [13] provide a detailed discussion of traditional recovery techniques, such as techniques presented by Jagadish et al. [20, 21], Eich et al. [10], and Levy et al. [25] for main-memory databases. All of them rely on logging and snapshotting, which are not needed in our recovery algorithm. In this family of traditional recovery techniques, ARIES [28] is without doubt the most well-known recovery technique.

The Shore-MT team focused on improving logging efficiency by eliminating log-related contention [22, 32, 33]. Besides, Cao et al. [3] propose a main-memory checkpoint recovery algorithm for frequently consistent applications. Furthermore, Lomet et al. [26] and Malviya et al. [27] investigate logical logging-based recovery techniques. Finally, Goetz et al. [15] present a technique called *Instant Recovery with Write-Ahead Logging*, that is based on on-demand single-page recovery. However, this technique does not apply to our architecture, as we do not use paging.

More recently, Yao et al. [46] studied the recovery cost of distributed main-memory database systems, investigating in particular the differential in recovery cost between transaction-level and tuple-level logging. Further, Wu et al. [44] proposed Pacman, a fast and parallel recovery mechanism for main-memory databases. Pacman assumes database transactions to be executed exclusively through stored procedures, which it analyzes at compile time to identify independent parts that can be executed in parallel during recovery.

6. CONCLUSION

The advent of SCM has enabled single-level database architectures to emerge. These store, access, and modify data directly in SCM, alleviating the traditional recovery bottleneck of main-memory databases, i.e., reloading data from storage to memory. Thus, the new recovery bottleneck for such systems is rebuilding DRAM-based data. In this paper, we addressed this bottleneck and presented a novel recovery technique that relies on: (1) a characterization of secondary data structures, (2) a ranking function for secondary data structures that takes into account the workload before and after crash, and (3) an adaptive resource allocation algorithm. We presented two benefit functions for secondary data structures: the first one considers indexes independently while the second one takes into account indexes interdependence. Our recovery algorithm aims at maximizing performance of the workload run in parallel with recovery. We have implemented adaptive recovery in our SCM-enabled database prototype, named SOFORT. Through a thorough experimental evaluation, we have shown that our approach significantly outperforms the existing synchronous and instant recovery approaches. In addition, the adaptive resource allocation algorithm brings a significant performance improvement over static resource allocation. We have demonstrated that our ranking functions adapt well to workload changes during recovery and significantly outperform rankings that do not take into consideration the recovery workload. Additionally, our adaptive recovery is robust and invariant with respect to SCM latency changes. Finally, our work paves the way for the development of next-generation database architectures on SCM.

7. REFERENCES

[1] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *ACM SIGMOD*, 2015.

[2] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4), 2016.

[3] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *ACM SIGMOD*, 2011.

[4] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5), 2015.

[5] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[6] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7), 2015.

[7] R. Colledge. *SQL Server 2008 Administration in Action*. Manning Publications, 2009.

[8] S. R. Dulloor. *Systems and Applications for Persistent Memory.* PhD thesis, Georgia Institute of Technology, 2016.

[9] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *ACM EuroSys*, 2014.

[10] M. H. Eich. Main Memory Database Recovery. In *ACM Proceedings of Fall Joint Computer Conference*, 1986.

[11] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *IEEE ICDE*, 2011.

[12] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: reducing transaction logging overhead with PCM. In *ACM CIKM*, 2011.

[13] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6), 1992.

[14] B. Govoreanu, G. Kar, Y. Chen, V. Paraschiv, S. Kubicek, A. Fantini, I. Radu, L. Goux, S. Clima, R. Degraeve, et al. $10\times$ 10nm 2 hf/hfo x crossbar resistive ram with excellent performance, reliability and low-energy operation. In *IEEE Electron Devices Meeting (IEDM)*, 2011.

[15] G. Graefe, W. Guy, and C. Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2016.

[16] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *PVLDB*, 4(2), 2010.

[17] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, et al. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *IEEE Electron Devices Meeting (IEDM)*, 2005.

[18] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4), 2014.

[19] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *In CIDR*, 2007.

[20] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A High Performance Main Memory Storage Manager. In *VLDB*, 1994.

[21] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from Main-Memory Lapses. In *VLDB*, 1993.

[22] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1-2), 2010.

[23] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *ACM SIGMOD*, 2015.

[24] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE/ACM MICRO*, 30(1), 2010.

[25] E. Levy and A. Silberschatz. Incremental Recovery in Main Memory Database Systems. *IEEE Trans. on Knowl. and Data Eng.*, 4(6), 1992.

[26] D. Lomet, K. Tzoumas, and M. Zwilling. Implementing performance competitive logical recovery. *PVLDB*, 4(7), 2011.

[27] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *IEEE ICDE*, 2014.

[28] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1), 1992.

[29] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In *ACM DaMoN*, 2014.

[30] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *ACM SIGMOD*, 2016.

[31] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. In *CIDR*, 2015.

[32] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2), 2010.

[33] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10), 2011.

[34] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVRAM Era. *PVLDB*, 7(2), 2013.

[35] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *IEEE/ACM MICRO*, 2009.

[36] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. *PVLDB*, 2(1), 2009.

[37] D. Schwalb, G. K. BK, M. Dreseler, S. Anusha, M. Faust, A. Hohl, T. Berning, G. Makkar, H. Plattner, and P. Deshmukh. Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory. In *International Conference on Database Systems for Advanced Applications*. Springer, 2016.

[38] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[39] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX FAST*, 2011.

[40] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[41] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[42] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1), 2009.

[43] R. S. Williams. How we found the missing memristor. *IEEE Spectrum*, 45(12), 2008.

[44] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *ACM SIGMOD*, 2017.

[45] J. Yang, Q. Wei, C. Wang, C. Chen, K. Yong, and B. He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, PP(99), 2015.

[46] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *ACM SIGMOD*, 2016.