Resource-Adaptive Query Execution with Paged Memory Management

Riki Otaki
University of Chicago
rotaki@uchicago.edu

Jun Hyuk Chang
University of Chicago
junhyukc@uchicago.edu

Charles Benello
University of Chicago
cmbenello@uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Goetz Graefe
Google
goetzg@google.com

ABSTRACT

Existing database systems typically employ demand driven resource allocation with static limits on the amount of resources allocated for query execution. However, in the cloud computing paradigm, where many different database workloads share the same physical resources, demand-driven approaches can lead to resource thrashing, while static limits can lead to under-utilization of resources. Under such circumstances, adaptively reallocating resources to operators in a query plan can improve resource utilization, performance, and reduce the operational costs. This paper explores the challenges of adaptive resource management, including estimating resource requirements of queries and dynamically redistributing resources to operators. We propose the use of the buffer pool pages for query execution to enable memory reallocation and query context switching seamlessly. Additionally, we propose an optimization technique to reduce the overhead of accessing the shared buffer pool memory.

1 INTRODUCTION

Cloud-based systems offer several advantages over on-premise systems, including cost-effectiveness, scalability, and elasticity. According to a recent market study, enterprise spending on Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) is projected to triple by 2027 compared to 2022 levels [3]. This significant increase is expected to drive greater demand for Database as a Service (DBaaS). As more database systems migrate to cloud-based platforms, efficiently executing diverse workloads on shared infrastructure has become a paramount challenge for cloud providers. Resource management strategies directly impact costs for both providers and tenants, making it a focal point of research in academia and industry alike [7, 8, 13, 14, 16, 17, 21, 39, 44, 45, 58]. From the cloud provider's perspective, resource management in cloud-based database systems is crucial for two main reasons.

First, the consolidation of multiple databases on shared infrastructure leads to a diversity of workloads within a single machine that is far greater than in traditional on-premise systems. These workloads have varying resource requirements in terms of time and space and are governed by different Service Level Agreements (SLAs). Relying solely on demand-driven resource allocation or static resource limits can be suboptimal for achieving high resource utilization and minimizing SLA violations across multiple tenants [44]. For instance, without dynamic or adaptive resource reallocation, a long-running query with a less stringent SLA might monopolize resources, causing a short-running query with a stricter SLA to wait unnecessarily. Such inefficiencies must be addressed to remain competitive, as providers offering better performance at lower costs will have a significant advantage.

Second, the rise of serverless computing [11, 51] has intensified the need for efficient resource management. The pay-as-you-go model of serverless computing is gaining popularity among applications with bursty or unpredictable workloads, as it allows them to pay only for the resources they actually use. This contrasts with serverful systems, where resources are reserved for set periods—typically calibrated for peak workloads—and remain underutilized much of the time [43, 51]. Consequently, resource requirements for cloud workloads have become more volatile and fine-grained, making agility in adapting to changing workloads and resource demands crucial.

To address these challenges, cloud providers need mechanisms to accurately estimate the resource requirements of workloads and swiftly meet these demands without adversely impacting performance. However, several challenges persist in achieving this goal.

First, accurate resource estimation is difficult. Predicting the resources a query will need is difficult because cardinality estimation is inherently complex [37], and determining the amount of resources to guarantee a tenant is complicated by the fact that resource allocation is tied to the user's willingness to pay and resource pricing can fluctuate based on demand [1, 40, 58]. Machine learning models trained on historical data can aid in predicting resource requirements. However, these models are not foolproof, as they often assume recurring patterns in workloads. In reality, workloads can be adhoc, bursty, and subject to change over time [47]. While large cloud providers have abundant historical data to train models [43, 56], relying exclusively on heavily trained ML models may not be the most efficient or cost-effective approach. Therefore, developing more adaptable and economical methods for modeling resource requirements is essential.

Second, agility in resource re-allocations is difficult to achieve as resources and workloads are not easily transferrable. When moving workloads between machines, migrations often suffer from states clean-ups and cold restarts [17, 21]. Memory fragmentation further complicates this process by making it challenging to move memory states across machines. In the case of suspending the current workload to run another, operations like query suspend and resume typically require re-executing the query or serializing a large amount of intermediate states to disk, leading to high latency overheads [12, 40, 41]. For reallocating resources from one workload to another, algorithms must continuously poll for resource availability and reconsider resource allocation decisions once they are made [15, 19, 54]. Determining how to resolve conflicting requests for resources among multiple tenants is also challenging, adding complexity to resource management.

Moreover, while cloud systems provide scalability by allowing the addition of more resources, the underlying database algorithms may introduce inherent bottlenecks—simply adding resources does not guarantee that the database will scale effectively. Shared resources such as locks, latches, and singleton objects like the buffer pool can become points of contention, limiting performance as concurrency increases [28, 30, 38]. High levels of concurrency and workload diversity exacerbate these issues, leading to inefficiencies that prevent the system from fully utilizing the available resources. Therefore, cloud database systems must employ scalable algorithms that minimize these bottlenecks to fully exploit their resources.

To address these challenges, we present a system design that effectively quantifies the benefits of resource allocation and dynamically reallocates resources among tenants through a cost-based approach, all without compromising scalability and performance. We propose utilizing buffer pool pages for both caching disk state and intermediate state generated during query execution. This allows for greater flexibility in dynamic resource adjustments such as resizing working memory and context-switching among queries.

In this paper, we focus on memory resources given it is a critical factor in database systems that affects other resource usages like CPU and I/O bandwidth. For instance, the amount of memory allocated to each query or operator determines the level of inter- or intra-query parallelism. Additionally, the availability of memory buffers influences data prefetching and caching, thereby affecting I/O bandwidth utilization.

This paper is organized as follows. First, it introduces query execution with buffer pool page memory, which enables memory adjustments and lightweight context-switching among queries. Then, it presents Logical ID with Physical Address Hinting (LIPAH), an optimization technique necessary for reducing contention in the shared buffer pool. Lastly, we show the preliminary evaluation of the proposed system design and optimization technique using the TPC-H benchmark [2].

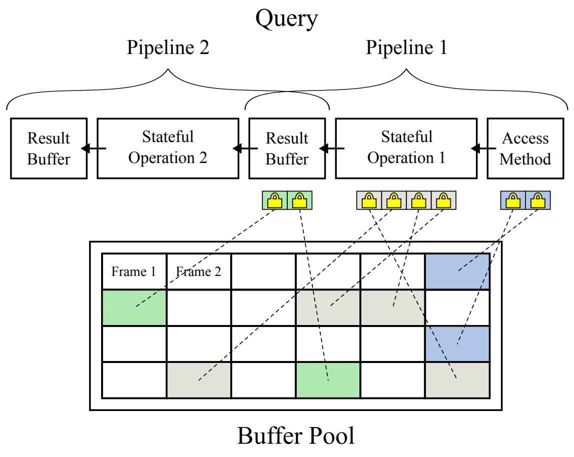


Figure 1: Buffer Pool Based Query Execution.

2 PAGED MEMORY FOR QUERIES

Traditional Database Management Systems (DBMS) typically segregate memory allocation for caching persistent table data from storing temporary intermediate computation results. For instance, in PostgreSQL, persistent data caches are stored in the shared buffer pool, while the working memory for query operators are allocated independently [5]. This segregation not only limits the utilization of memory resources [32, 33] but also the flexibility of memory allocation adjustments during query execution.

Consider a scenario where it is necessary to reduce the memory allocated for a query due to other high-priority queries. With the traditional approach, the operator's intermediate results are often allocated on the heap memory using memory allocation libraries such as `malloc`. However, these allocations are not guaranteed to be contiguous and may be interleaved with other allocations. Consequently, spilling these states to disk incurs overhead due to the need for serialization of data and abrupt I/O operations to the storage system [40].

We propose that for flexible memory allocation, both file caching and query operator memory should be managed using pages within the buffer pool. Allocating memory in buffer pool pages allows each memory consumer to resize memory allocations by simply pin and unpin pages, avoiding the overhead of serialization and deserialization and enabling efficient exchange of memory between memory consumers. Moreover, the buffer pool can evict pages to disk lazily when under memory pressure, so that it can minimize the overhead of spilling intermediate states to disk.

For instance, suppose a query's working memory is allocated as pages in the buffer pool. During execution, the query can pin the pages in the buffer pool to prevent eviction to disk so they can be used for intermediate results. When the query needs to be suspended due to higher priority queries, it can release all of the pins, allowing the buffer pool to utilize the pages for other queries. When the query is resumed, it can request the pages from the buffer pool and continue execution. This approach avoids workload spikes on the storage system and provides more predictable performance during query preemption.

To our knowledge, DuckDB [50] is one of the first systems to employ buffer pool pages for query execution [31, 32], focusing on gracefully handling the transition between in-memory and disk-based execution in locally running systems. However, DuckDB is built as an embeddable library optimized for local environments, whereas our system assumes a database server that handles requests from many clients. In such multi-client, multi-tenant environments, memory resources are shared among multiple queries and clients, necessitating different design considerations for resource management. To address these challenges, we propose the use of cost-aware memory allocation, adaptive memory resizing, and scalable buffer pool management specifically tailored for cloud-based database systems, as described in the following sections.

The query model of our system, as illustrated in Figure 1, showcases how the buffer pool is used for both file caches and intermediate query results. Our buffer pool implements a standard model with fixed-size pages and a centralized page-to-frame mapping table to track and locate pages. When the buffer pool is full, it samples a fixed number of frames and evicts the one with the lowest score based on a page replacement policy. Each frame has a read-write latch to prevent concurrent access and pin pages in memory, ensuring they are not evicted while in use.

The query execution engine breaks a query plan into a graph of pipelines, each consisting of a sequence of operators ending with a stateful operator [36, 55]. Operators within a pipeline can stream data, but the stateful operator must read all input data before outputting results. The first pipeline, illustrated in Figure 1, involves a scan operator that reads data by latching frames in the buffer pool. The stateful operator, such as a sort operator, requires scratch space and will allocate frames from the buffer pool as needed. After executing the stateful operator, the results are written to the result buffer. Once the first pipeline completes, latches are released, and dependent pipelines are notified. Ready pipelines are queued and executed sequentially until all are processed, and the final result buffer is returned to the caller.

There are several overheads associated with paged memory allocation for query execution.

Serialization/Deserialization Overhead: With paged memory allocation, the query operator must maintain intermediate states in a page-oriented data structure. For example, a hash-aggregate operator stores aggregated values in a paged hash table instead of a non-paged one. When processing a new tuple, the operator deserializes the bytes from the page into a computable tuple, updates the aggregate value, and then serializes it back to the page. This overhead can be avoided by using zero-copy techniques to access the record directly from the page but this requires careful management of the memory layout especially when the record contains variable-length fields.

Data Access Overhead: Accessing an element in a paged memory structure incurs more overhead than in non-paged memory, where elements are accessed directly via pointers. In paged structures, one must first access the page in the buffer pool and then the element within the page. This involves accessing the centralized page-to-frame mapping table, often causing overhead due to contention. Additionally, accessing elements in paged memory (e.g., slotted pages) requires finding the element's offset in the slot array and then accessing the element, which can lead to cache misses and extra memory accesses.

Page Organization Overhead: In order to maintain a high fill factor of pages or to maintain a constraint of the paged data structure, pages may need to be compacted, merged, or split occasionally. These operations can be costly due to the need to move data between pages and update metadata. Finding a space for a new element in a page may trigger a page organization operation, which can lead to additional overhead.

Techniques have been devised to mitigate these overheads and improve the efficiency of using paged memory allocation in disk-based DBMSs. *Poor man's normalized keys* have been proposed to reduce the key access overhead when running binary searches on sorted pages in a B-tree index [27]. Adding hints in the page header can further reduce the latency by limiting the range of binary searches [6]. DuckDB separates rows into fixed size components and variable size components and place them in separate pages to accelerate the access of fixed size rows [32]. These techniques can be adapted to our system to reduce the overhead of paged memory allocation.

We are actively exploring more techniques to mitigate these overheads and improve the efficiency of paged memory allocation for query execution. Our goal is to provide flexibility in memory allocation by consolidating and managing memory at the granularity of pages, while achieving performance competitive with non-paged memory allocation. This approach will enable more efficient resource management, as memory can be easily transferred between different consumers—such as query operators and file caches.

2.1 Cost-Aware Memory Allocation

The core question of the buffer pool based query execution is how to decide the number of pages allocated for a stateful operator in a pipeline. A stateful operation, such as sorting, aggregation, and hash table creation, requires a certain amount of scratch space to run the algorithm. The amount of scratch space does not change the correctness of the algorithm but affects the performance of the algorithms.

We first describe a motivating example to illustrate the problem. Suppose that the buffer pool has 1000 frames, constantly issuing 100 IOPS to the disk. Then the sort operator requests the buffer pool for some amount of frames for working memory. Consider the following two cases. In the first case, the buffer pool gives 300 frames to the sort operator, and the sort can run the algorithm with one pass of the input data. In the second case, the buffer pool gives 50 frames to the sort

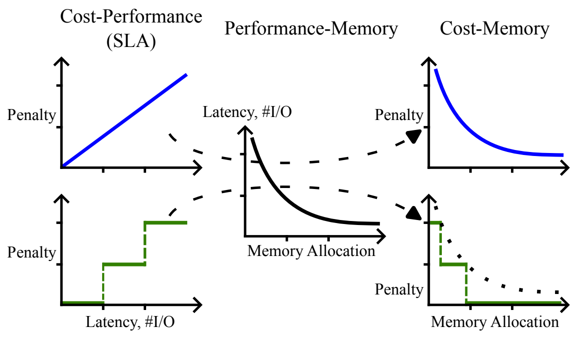


Figure 2: Deriving Cost-Memory Graph from the SLA and Performance-Memory Model.

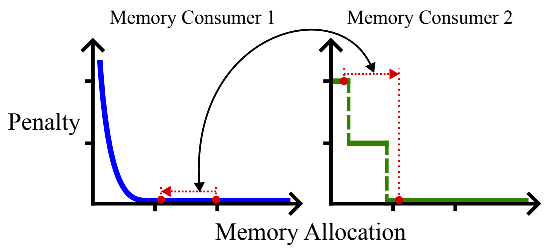
operator, and the sort generates multiple runs that have to be merged in the subsequent steps.

From the operator's perspective, the first case is more desirable than the second case because of the lower latency. However, the buffer pool may suffer from higher IOPS in the first case due to having less free frames to cache pages for other queries. In the second case, the sort operator may suffer from higher latency due to the merge phase, but the buffer pool may not suffer from increased IOPS.

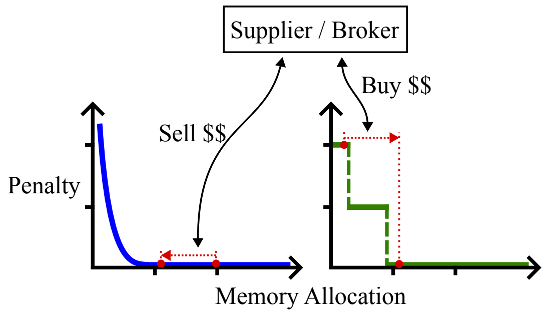
Our goal is to quantify the benefit of memory to each memory consumer and decide which memory consumer should be granted memory or should release memory. This requires an economic analysis of the value of memory to each memory consumer such as query operators and file caches. By incorporating microeconomic principles into memory allocation, we can guide memory allocation decisions based on the cost function of each memory consumer.

Introducing microeconomic principles into memory allocation decisions involves two steps. First, we need to model the relationship between memory and cost for each workload, effectively quantifying how additional memory impacts the cost of the workload. Second, we need to design a mechanism to allocate and release memory based on this cost function, ensuring that memory resources are distributed in a way that maximizes their overall value.

Modeling Cost-Memory Relationship: To model the relationship between memory and cost, we need to define cost in the context of our system. Here, we suppose that the cost of a workload can be quantified by the monetary penalty of violating the SLA. The SLA can be defined in terms of query performance metrics, such as query latency or the number of I/O operations. Although many commercial systems provide SLAs based on resource availability rather than query performance [43], we believe that customers will benefit greatly from performance-based SLAs, as they directly affect user experience. An example of a performance-based SLA is provided by Azure Cosmos DB, a key-value store that guarantees an SLA of less than 10ms for the 99th percentile of point read and write operations [10].



(a) Exchange-based memory allocation. Memory is exchanged between two consumers if it benefits both consumers or reduces the global cost.



(b) Price-based memory allocation. Prices of memory units are set by the supplier or broker based on the demand and supply. Memory consumers buy and sell memory based on their budget and needs.

Figure 3: Economic models for memory allocation.

To derive the cost-memory relationship from the SLA, the effect of memory size on performance must be known. This can be computed using theoretical models or obtained via empirical measurements [14]. For instance, the number of I/O operations required for a naive external sort algorithm with M memory blocks can be analytically computed as

$$b_r \left(2 \left\lceil \log_{\lfloor M/b_b \rfloor - 1} (b_r/M) \right\rceil + 1 \right)$$

where b_r is the total number of blocks containing records of relation r and b_b are read and written at a time [52]. As memory allocation M increases, the logarithmic term decreases because the base of the logarithm increases ($\lfloor M/b_b \rfloor - 1$ increases). This results in fewer merge passes needed to sort the data, thereby reducing the number of I/O operations. Using the performance-memory model, we can derive the cost-memory graph (see Figure 2), which represents the penalty incurred by the workload as memory size changes. The structure of this graph can vary depending on the SLA and the performance-memory model; for example, if the SLA is defined as a step function, the cost-memory graph might also exhibit a step function structure. This insight can guide memory allocation decisions, as certain memory size ranges may be superfluous for the workload.

Microeconomic Mechanisms for Memory Allocation: There are two major approaches to incorporating microeconomic mechanisms into memory allocation decisions: exchange based models and pricing based models [22]. Figure 3 illustrates using these models to reallocate memory between two memory consumers using the cost-memory graph derived in the previous step.

In exchange based models, each memory consumer is endowed with some initial memory allocation and can selfishly or cooperatively exchange resources to optimize their own cost or the system's global cost. In the selfish approach, memory consumers will only exchange memory if it does not harm their own cost. This behavior typically leads to *Pareto-optimal* solutions, where no consumer can benefit from exchanging memory without making another consumer worse off. In the cooperative approach, memory consumers negotiate and exchange resources if the system's global penalty can be reduced. For example, if one consumer's marginal cost of memory–that is, the additional penalty incurred per unit of memory shortage–is higher than another's, it is is beneficial for the system to reallocate memory from the consumer with the lower marginal cost to the one with the higher marginal cost. This reallocation reduces the overall penalty, thus optimizing the system's performance.

In pricing-based models, memory units are allocated based on assigned prices. Each memory consumer is charged for the resources they consume, with prices reflecting the scarcity and demand for those resources. Memory consumers make decisions to optimize their cost while considering these resource prices. This approach can lead to efficient resource allocation by providing incentives for consumers to use resources judiciously. However, distributing budgets and setting prices can often be challenging due to the variability of the resource capacity and workload demands.

Auction-based mechanisms can help mitigate these challenges and has been explored in the context of distributing resources between hash join operators [18], choosing execution sites for distributed query processing [53], and reallocating memory between databases in multi-tenant systems [7]. In these mechanisms, memory consumers bid for resources based on their cost functions and budgets. The system then allocates memory based on the bids, ensuring that more memory is allocated to consumers who value it more. Often, these auctions require a centralized supplier or broker to manage the bidding process, which can introduce additional overhead due to communication and coordination costs. This overhead must be carefully managed to ensure that the benefits of the auction mechanism outweigh it.

Several research questions remain open in this area. First, designing efficient communication protocols between memory consumers and suppliers is crucial. Frequent communication leads to agile resource allocation, but it can also introduce overhead due to message passing and synchronization. Second, determining rational pricing mechanisms for suppliers is essential in pricing-based models. Prices must be set to attract memory consumers while ensuring that the system remains profitable in terms of resource utilization and cost. Finally, establishing protocols for price guarantees to memory consumers is critical. Delays in price information or surges in demand can lead to price fluctuations, causing memory consumers to make suboptimal decisions. These challenges are ongoing research questions that are critical to our efforts.

2.2 Adaptive Memory Resizing

A key challenge in buffer pool-based query execution is resizing memory allocations for file caches and query operators in response to changing workloads. Resizing memory cached for files is straightforward. The buffer pool manager can evict file cache pages to disk when memory consumption exceeds some threshold. Resizing memory used for operators' working memory is more challenging.

Similar to file caches, the buffer pool manager can evict pages of the operators' working memory in the background without informing the operator itself. However, an operator's working memory will likely be accessed frequently during execution. For example, a sort operator's sort buffer will be accessed when the data is first loaded, when the data is read to sort, and when the data is read to return the sorted result. Evicting these pages to disk in the middle of the sort operation without informing the sort operator would be inefficient because the operator would have to read the pages from disk again to continue the sort operation.

Two approaches can be considered to address this issue.
Memory-Aware Operator Algorithms: The first approach is to make each memory consumer aware of which pages are in the buffer pool. This can be achieved by pinning the pages so the buffer pool cannot evict them. After the memory allocation strategy is decided based on the cost-memory relationship, each memory consumer can decide whether to request more memory or release some of the pinned working memory. This can be done without terminating the operator's execution if we adopt memory-adaptive algorithms [18, 49] that can resize memory allocation during execution.
Memory-Oblivious Operator Algorithms: The second approach is to optimize algorithms for RAM-oblivious environments. Just as cache-oblivious algorithms [23] optimize cache use without relying on the cache size, operator algorithms can be designed to be elastic to the available memory size. In this way, even if the pages of the working memory are evicted to disk during execution, the operator can continue executing without significant performance degradation.

We are investigating both algorithms to identify the most effective methods for dynamically resizing memory allocations for query operators in response to changing workloads. By integrating these approaches into the query execution runtime, the system can adjust memory allocations and perform context-switching as needed, ensuring efficient and effective utilization of memory resources. Successful implementation of these methods requires careful consideration of several critical factors: prioritizing which pages to evict, determining the optimal frequency of memory resizing, and ensuring that memory

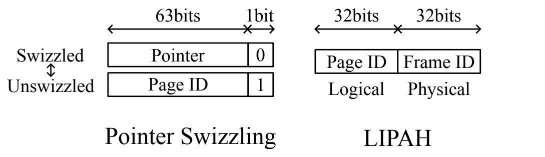


Figure 4: Comparison of Pointer Swizzling and LIPAH.

consumers can adapt to changing memory allocations without significant performance degradation.

3 LIPAH

Our system shares the buffer pool among multiple query operators and file caches. The buffer pool manager tracks pages in the buffer pool. However, the centralized page-to-frame mapping table can become a point of contention during concurrent accesses by multiple query operators and file caches. To reduce the contention and overhead, we propose a technique called *Logical ID with Physical Address Hinting* (LIPAH). This technique is inspired by the works of pointer swizzling [24, 28, 38, 48], but is more robust to arbitrary data structures and easier to implement.

Figure 4 shows the structure of LIPAH. The key idea of LIPAH is to use a fat-pointer, which is a pair of page ID and the frame ID as a reference to the page in the buffer pool. Here, page ID is a logical ID unique to the page and frame ID is the physical address of the page in the buffer pool. The page is accessed using frame ID first without consulting the centralized page-to-frame mapping table. If the frame does not contain the page with the page ID, the buffer pool manager will consult the mapping table to find the frame.

This is similar to the existing pointer swizzling techniques. Existing pointer swizzling techniques required *swizzling* the logical ID of the page to the physical address when the page is loaded and *un-swizzling* the physical address to the logical ID when the page is offloaded to a secondary storage. There are two limitations to this approach.

First, un-swizzling all the physical references to the page is necessary when the page is offloaded to a secondary storage. This is a costly operation if there are many references to the page, as the buffer pool manager must traverse all the references to the page to un-swizzle the physical addresses.

Second, the page subject to eviction must not contain any swizzled references. Otherwise, when the page is loaded back to the buffer pool, the swizzled references might point to invalid physical addresses. This necessitates a dense connection between the buffer pool manager and the access method algorithms to ensure the buffer pool can find an evictable page.

Existing works have addressed these limitations by employing B-tree structures that limit each page to at most one incoming edge [28, 38]. By evicting child pages before their parent pages, they ensure that no swizzled pointers are written to disk. Write-optimized B-trees [25, 26] are examples of such B-tree structures. This design keeps the number of incoming edges to a page minimal, allowing page migrations, splits, and merges to be performed with minimal overhead. However, these structures do not support sibling page references, which are important for scan operations. Consequently, to support arbitrary data structures, a more flexible approach is required.

In LIPAH, a reference to a page is a pair of the logical ID and the physical address of the page. This solves the two limitations of the existing techniques because it allows page references to have an invalid physical address. If the physical address is invalid, the buffer pool manager can still use the logical ID to find the object in the buffer pool.

In LIPAH, accessing an object in the buffer pool entails the following steps. First, the buffer pool manager looks up the frame specified by the physical address. It then checks if the logical ID of the page in the frame matches the logical ID in the pair. If so, the buffer pool manager returns the page in the frame. We call this a fast-path access. This is the same as the existing pointer swizzling techniques, where the physical address is used to find the page in the buffer pool. If the logical ID does not match and fast-path access fails, the buffer pool manager will consult the translation table to find the frame that contains the page with the logical ID. After finding the frame (hit) or loading the page from the secondary storage (miss), the access method can update the physical address in the pair to the new address of the page. We call this a slow-path access. In this way, the physical address is only used as a hint to find the page in the buffer pool, and there is always a fallback mechanism to find the page using the logical ID.

Implementation-wise, the logical ID is a page ID and the physical address is a frame ID. Both of them are 4-byte unsigned integers. Initially, the frame ID is set to a maximum value. The buffer pool will ignore the fast-path access if the requested frame ID is larger than the number of frames it has and will use the logical ID to find the page in the buffer pool. After a slow-path access, the access method typically updates the page's physical address in the parent page used to find the page. This is opportunistically done by trying to acquire a write-latch on the parent page and updating the physical address if the latch is available.

Using LIPAH, the buffer pool manager can express graph data structures with circular references. The existing work on pointer swizzling had difficulty in expressing graph data structures because pages that are subject to eviction must have zero swizzled references [34, 35, 38]. If two pages in a graph have a swizzled reference to each other, the buffer pool manager can not evict either of the pages. In LIPAH, the buffer pool manager's eviction policy is not affected by the data structure expressed on the buffer pool. The buffer pool manager can evict any page in the buffer pool as long as the page is not being accessed by the access method, which can be known by the status of the frame latch. LIPAH can also be utilized in representations where graphs are stored as sets of edges instead of vertices with pointers, avoiding the overhead of swizzling and un-swizzling pointers.

Another work that addresses the problem of pointer swizzling is *vmcache* [35]. The primary difference between vmcache and LIPAH lies in their approach to indirection: vmcache aims to eliminate the indirection of page-to-frame mapping altogether, effectively removing this layer to improve performance. In contrast, LIPAH retains the indirection layer but utilizes hints to bypass it. Moreover, while vmcache is specific to the buffer pool, LIPAH's flexibility allows it to be applied in other areas. For example, LIPAH can be used in secondary indexes to enhance performance.

By integrating LIPAH into secondary indexes that maintain references to the primary index, the index can store not only the primary key as the logical ID but also the physical address of the record as a hint. This reduces the overhead of traversing the primary index for each lookup. Oracle Database employs a similar technique to accelerate lookups in the secondary index by storing the physical address of the record within it [29]. By applying LIPAH's hint-based approach, we can further optimize secondary index performance. We plan to explore more applications of LIPAH in future work.

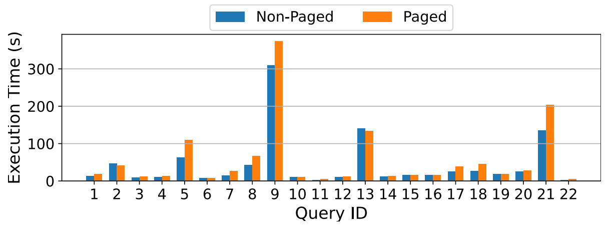
One limitation of LIPAH is the memory overhead introduced by the use of fat-pointers. In our implementation, each fat-pointer occupies 8 bytes—4 bytes more than a page ID alone— due to the inclusion of an additional frame ID. However, these fat-pointers are stored only in the internal nodes of the access method's data structures. In B+-tree organized files, internal nodes store keys and pointers to child nodes, while leaf nodes store the keys and actual values. Since the number of internal nodes is typically #fan-out times smaller than the number of leaf nodes, overhead introduced by fat-pointers is small relative to the total memory consumption of the system. Moreover, we believe that the benefits of LIPAH in reducing contention in the buffer pool manager outweigh this modest overhead.

4 PRELIMINARY EVALUATION

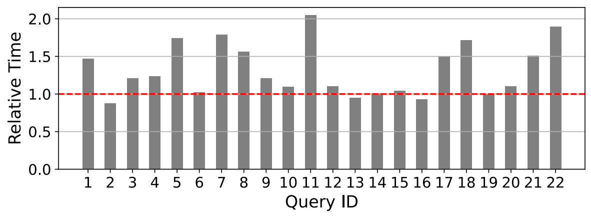
All the experiments were run on a system configured with dual-socket Intel® Xeon® Silver 4116 CPUs. Each socket hosts 12 physical cores operating at a clock speed of 2.10 GHz, with hyper-threading enabled, allowing for 2 threads per core, with 192 GB of RAM. Pages are sized at 256 KB.

The database system used in our experiments is implemented in Rust. It currently employs a row-based storage model, which is not optimized for executing OLAP queries, and lacks a cost-based query optimizer. As a result, its performance is not comparable to that of a mature database system. However, our experiments are designed to demonstrate the relative performance differences when enabling certain features within the same system, rather than comparing it to other database systems. This approach provides insight into the potential performance differences that could be achieved if these features were applied to a mature database system.

We first compared the performance of TPC-H queries with and without page based memory management. In a non-paged system, all the data (persistent and intermediate) is stored using Rust's native container types (i.e., `Vec` and `HashMap`). We implemented the equivalent data structures using the buffer pool in the paged system and used them to store all the data in



(a) Query Latency.



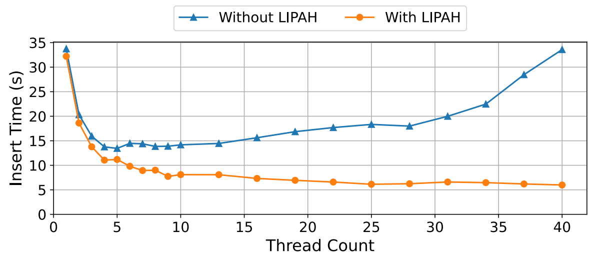
(b) Relative Query Latency (Paged / Non-Paged).

Figure 5: TPC-H (SF1) Query Latencies Comparing Paged and Non-Paged Allocations.

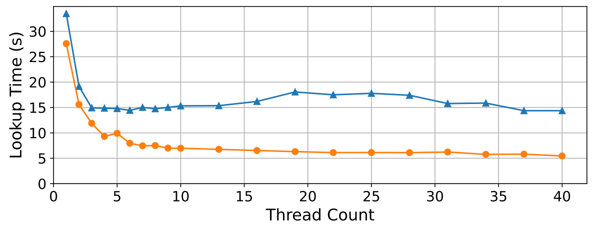
the system. These data structures fully utilize LIPAH to access the pages in the buffer pool. Hence, the page access will be a fast-path access except for the first access to the page if the query is run fully in memory. The query plans and the set of pipelines are the same in both systems.

We assessed the performance of TPC-H queries with a scale factor of 1. The buffer pool was configured with 600 thousand frames, corresponding to approximately 150 GB of memory. We chose a scale factor of 1 to ensure that all data, including the intermediate states of queries, could fit within the buffer pool. Prior to running the actual queries, we executed the same query once as a warm-up to guarantee that the data was fully loaded into the buffer pool. The query was executed with 1 thread and the maximum working memory of blocking operators was set to 600 pages. We measured the latency of each query three times and calculated the average latency.

Figure 5a shows the latency of TPC-H queries with and without page-based memory management. Figure 5b shows the relative latency of paged system to non-paged system. Out of 22 queries, 7 queries were more than 1.5 times slower in in paged system than non-paged system. We identified that those queries with identical performance require evaluating complex expressions. For example, queries 2, 9, 13, 14, 16, and 20 contain `LIKE` expressions that require evaluating regular expressions. This dominated the execution time, making the overhead of the paged system negligible. For queries with shorter execution time such as query 11 or 22, they are more sensitive to the overhead of the paged system. In the paged system, we have not fully optimized the execution cost beyond using LIPAH. We believe that with further optimization, such as using more efficient data layout, enabling computation over serialized data, and reducing slot access overhead by allocating slot metadata and data in a contiguous memory region, we



(a) Insertion Latency.



(b) Lookup Latency.

Figure 6: Latencies of Insertion and Lookup Operations in Hash Index with and without LIPAH.

can reduce the overhead of the paged system and improve its performance.

Next, we assessed the performance of LIPAH by integrating it into a hash index on the buffer pool, comparing it to a standard hash index without LIPAH. Both indexes used paged memory management. The hash index employed a chaining mechanism for collision handling. Inserting a key-value pair involved hashing the key to locate the appropriate bucket, and if full, linking a new page to it. For lookups, we hashed the key to access the bucket and traversed the linked-list to find the pair. We measured the latency for these operations using keys of 100 bytes and values between 50 and 100 bytes. We processed 10 million unique key-value pairs, inserting them and conducting lookups in a random sequence. The entire dataset was contained within the buffer pool, eliminating disk I/O during the experiment. We ran the experiment three times and calculated the average latency.

Figure 6 illustrates the latency associated with inserting and retrieving a key-value pair in a hash index, comparing operations with and without the LIPAH. The data shows that LIPAH significantly enhances the speed of both insertion and lookup tasks compared to a conventional hash index. This efficiency gain arises because LIPAH enables direct access to frames within the buffer pool, bypassing the centralized mapping table typically required. During the experiment, we configured the hash index with 1,024 buckets and noted that after inserting 10 million key-value pairs, approximately 7,500 pages were utilized. Consequently, the average chain length was approximately 7.3, implying that each key lookup accessed on average 3.65 pages. Hence, LIPAH resulted in 3.65 fewer accesses of the page-to-frame mapping table per key lookup, significantly enhancing performance.

The normal hash index exhibits significant performance degradation with an increasing number of threads during insertion processes, as insert operation necessitates exclusive latching of the page-to-frame mapping when creating a new page into the buffer pool, blocking other threads. This latching leads to poor scalability in performance as thread count rises. In contrast, LIPAH scales effectively with increasing the thread count. This scalability is attributed to the design where page-to-frame mapping is not required for traversing the pages in the bucket.

For lookup operations, although more efficient than insertions, the normal hash index does not scale effectively with increasing threads. The operation secures a shared-latch on the page-to-frame mapping to ensure stability during page identification, allowing concurrent access. However, this shared-latch still dirties the cache and generates contention, degrading performance as thread numbers grow. LIPAH, devoid of latching the page-to-frame mapping, does not suffer from performance degradation, maintaining its efficiency across varying thread counts.

5 RELATED WORK

Given the scope of related work on adaptive query processing and multi-tenancy for database systems, we point the interested reader to foundational surveys on these topics [20, 43]. In this section we limit our discussion to dynamic memory management and adaptive memory allocation.

Dynamic Memory Management: In the 2000s, commercial database systems began incorporating advanced dynamic memory management techniques to adaptively respond to changing system demands, enhancing performance and reducing the need for manual tuning. For instance, Oracle's Automatic Memory Management (AMM) dynamically adjusts memory allocations based on real-time workload analysis [15]. Similarly, IBM DB2 introduced the Self-Tuning Memory Manager (STMM) [54], and Microsoft's SQL Server implemented resource monitoring systems [46] and memory broker systems [4] to optimize memory usage. While these adaptive mechanisms are effective, they often rely on complex algorithms that increase system complexity and computational overhead. Our approach simplifies memory management by leveraging paged memory, reducing overhead and streamlining spill handling. This offers a more straightforward and efficient solution that complements existing techniques with less complexity.

Quantifying the Importance of Memory: Several prior works have proposed quantifying the importance of memory to facilitate resource reallocations. Davison and Graefe proposed a resource broker system that allows query operators to exchange the resources between queries running in parallel [19]. Their implementation was based on *Return On Consumption* (ROC) model [57], which measures the effectiveness of additional memory consumption on response times improvement. Curino et al. proposed a technique to identify the active working set size, to build models for expected I/O given a memory allocation for consolidation [13]. Tang et al. [55] and Nagel et al. [42] developed mechanisms to identify query state to cache for effective reuse in queries. Brown et al. [9] and SQLVM [44, 45] defined the importance of a memory page based on the buffer pool hit rate. Arora et al. proposed a resource broker system that can balance the amount of resources allocated to each database system in a multi-tenant environment [7]. Their system uses *Value of Memory* (VOM) as the currency system, which is a multiple of *System Time Saved* (STS) and number of accesses to the memory per unit time. Collectively, these prior works demonstrate that currency systems can effectively quantify the importance of memory for resource reallocations. Building upon this foundation, we extend these concepts by employing memory pages as the fundamental unit of allocation. This design simplifies the exchange of resources between queries and file caches, reducing the overhead associated with resizing memory allocations.

6 CONCLUSION

Adaptively reallocating resources is a vital aspect of cloud database systems due to the inherent fluctuations in workload and resource availability compared to on-premise systems. This paper proposes query execution with paged memory that incorporates dynamic resource adjustment and lightweight context switching. This framework allows cloud providers to allocate resources more effectively to critical workloads based on a cost metric, enhancing overall system performance. Additionally, the proposed LIPAH optimization technique significantly accelerates page access within the buffer pool. We believe this framework will be the foundation for resource-efficient and adaptive cloud database systems.

ACKNOWLEDGMENTS

This work was in part supported by NSF Award IIS-2048088 and a Google Data Analytics and Insights (DANI) Award. The authors would also like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] [n. d.]. Spot Instances - Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.

[2] 1999. TPC-H Homepage. https://www.tpc.org/tpch/.

[3] 2023. Magic Quadrant for Cloud Database Management Systems. https://www.gartner.com/interactive/mq/5041631.

[4] 2023. Sys.Dm_os_memory_brokers (Transact-SQL) - SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-os-memory-brokers-transact-sql?view=sql-server-ver16.

[5] The PostgreSQL Global Development Group . 2024. 19.4. Resource Consumption. https://www.postgresql.org/docs/17/runtime-config-resource.html.

[6] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW*, Vol. P-331. 259–281.

[7] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4202–4215.

[8] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. In *VLDB*.

[9] Kurt P. Brown, Michael J. Carey, and Miron Livny. 1993. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB*. 328–341.

[10] Mark Brown. 2024. Consistency Level Choices - Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[11] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Communications of The Acm* 62, 12 (Nov. 2019), 44–54.

[12] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. 2007. Query Suspend and Resume. In *SIGMOD*. 557–568.

[13] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD*. 313–324.

[14] Carlo Curino, Eugene Wu, Evan P C Jones, Sam Madden, Raluca Ada Popa, Hari Balakrishnan, Nirmesh Malviya, and Nickolai Zeldovich. 2011. Relational Cloud: A Database-as-a-Service for the Cloud. In *CIDR*.

[15] Benoît Dageville and Mohamed Zait. 2002. SQL Memory Management in Oracle9i. In *VLDB*. 962–973.

[16] Sudipto Das, Vivek Narasayya, Feng Li, and Manoj Syamala. 2013. CPU Sharing Techniques for Performance Isolation in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* (Sept. 2013), 12.

[17] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2010. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *UCSB Computer Science Technical Report* 9 (2010).

[18] Diane L. Davison and Goetz Graefe. 1994. Memory-Contention Responsive Hash Joins. In *VLDB*. 379–390.

[19] Diane L. Davison and Goetz Graefe. 1995. Dynamic Resource Brokering for Multi-User Query Execution. In *SIGMOD*. 281–292.

[20] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (2007), 1–140.

[21] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*. 301–312.

[22] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini. 1996. Economic Models for Allocating Resources in Computer Systems. In *Market-Based Control: A Paradigm for Distributed Resource Allocation*. 156–183.

[23] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1, Article 4 (Jan. 2012).

[24] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.).

[25] Goetz Graefe. 2004. Write-Optimized b-Trees. In *VLDB*. 672–683.

[26] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. 2012. Foster B-Trees. *ACM Transactions on Database Systems* 37, 3, Article 17 (Sept. 2012).

[27] Goetz Graefe and Per-Åke Larson. 2001. B-Tree Indexes and CPU Caches. In *ICDE*. 349–358.

[28] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 37–48.

[29] Janis Greenberg, Sundeep Abraham, Shashaanka Agrawal, Geeta Arora, Eric Belden, Chandrasekharan Iyer, Geoff Lee, Anand Manikutty, Valarie Moore, Magdi Morsi, Helen Yeh, Adiel Yoaz, Qin Yu, and Sylaja Kannan. 2024. Design Considerations for REFs. https://docs.oracle.com/en/database/oracle/oracle-database/23/adobj/design-considerations-for-REFs.html#GUID-79DD95A2-3080-47BC-95FB-7FD42D1E1BBF.

[30] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *SIGMOD*. 981–992.

[31] Laurens Kuiper. 2024. No Memory? No Problem. External Aggregation in DuckDB. https://duckdb.org/2024/03/29/external-aggregation.html.

[32] Laurens Kuiper, Peter Boncz, and Hannes Muhleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *ICDE*.

[33] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2023. Cooperative Memory Management for Table and Temporary Data. In *SiMoD*. Article 2.

[34] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. In *VLDB*.

[35] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1, Article 7 (May 2023).

[36] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*. 743–754.

[37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215.

[38] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[39] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (May 2021), 1606–1612.

[40] Rui Liu, Jun Hyuk Chang, Riki Otaki, Zhe Heng Eng, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. 2024. Towards Resource-adaptive Query Execution in Cloud Native Databases. In *CIDR*.

[41] Rui Liu, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. 2024. Riveter: Adaptive Query Suspension and Resumption Framework for Cloud Native Databases. In *ICDE*. 3975–3988.

[42] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. 2013. Recycling in Pipelined Query Evaluation. In *ICDE*. 338–349.

[43] Vivek Narasayya and Surajit Chaudhuri. 2021. Cloud Data Services: Workloads, Architectures and Multi-Tenancy. *Found. Trends Databases* 10, 1 (May 2021), 1–107.

[44] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*.

[45] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 726–737.

[46] D. Narayanan, E. Thereska, and A. Ailamaki. 2005. Continuous Resource Monitoring for Self-Predicting DBMS. In *MASCOTS*. 239–248.

[47] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan (Murali) Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *SIGMOD/PODS*.

[48] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with in-Memory Performance. In *CIDR*.

[49] HweeHwa Pang, Michael J. Carey, and Miron Livny. 1993. Memory-Adaptive External Sorting. In *VLDB*. 618–629.

[50] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*. 1981–1984.

[51] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing Is and Should Become: The next Phase of Cloud Computing. *Communications of The Acm* 64, 5 (April 2021), 76–84.

[52] A. Silberschatz, H.F. Korth, and S. Sudarshan. 2020. *Database System Concepts*.

[53] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. 1994. An Economic Paradigm for Query Processing and Data Migration in Mariposa. In *PDIS*. 58–67.

[54] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-Tuning Memory in DB2. In *VLDB*. 1081–1092.

[55] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent Query Processing. *Proc. VLDB Endow.* 12, 11 (July 2019), 1427–1441.

[56] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *SIGMOD*. 280–294.

[57] Philip S. Yu and Douglas W. Cornell. 1993. Buffer Management Based on Return on Consumption in a Multi-Query Environment. *Proc. VLDB Endow.* 2, 1 (Jan. 1993), 1–37.

[58] Huanchen Zhang, Yihao Liu, and Jiaqi Yan. 2024. Cost-Intelligent Data Analytics in the Cloud. In *CIDR*.