Data Management for Data Science Towards Embedded Analytics

Mark Raasveldt
CWI Amsterdam
m.raasveldt@cwi.nl

Hannes Mühleisen
CWI Amsterdam
hannes@cwi.nl

ABSTRACT

The rise of Data Science has caused an influx of new users in need of data management solutions. However, instead of utilizing existing RDBMS solutions they are opting to use a stack of independent solutions for data storage and processing glued together by scripting languages. This is not because they do not need the functionality that an integrated RDBMS provides, but rather because existing RDBMS implementations do not cater to their use case. To solve these issues, we propose a new class of data management systems: embedded analytical systems. These systems are tightly integrated with analytical tools, and provide fast and efficient access to the data stored within them. In this work, we describe the unique challenges and opportunities w.r.t workloads, resilience and cooperation that are faced by this new class of systems and the steps we have taken towards addressing them in the DuckDB system.

1. INTRODUCTION

The rise of Data Science has considerably increased the complexity of data analysis tasks. As a result, these tasks require more advanced tools than standard SQL queries [11]. At the same time, computer capabilities have improved tremendously, which allows increasingly large data sets to be processed locally on personal computers. Due to these factors, data scientists are using scripting languages (e.g. Python or R) to run medium-sized analyses on their personal computers [15] instead of on large dedicated servers.

These data scientists are still in need of data management solutions, however, the data management community has fallen short of providing them. Data warehouses, the traditional solution to analytical workloads, are ill-equipped to support this use case. They are designed to run on server-grade hardware and be set up and maintained by experts. The standard SQL interfaces that they provide are insufficient to perform complex analysis tasks, and integrating these data warehouses with local tools is cumbersome and slow due to inefficient client protocols [21].

Due to the lack of systems that effectively support the local data analysis use case, a plethora of database alternatives have sprung up. For example, in Python and R basic data management operators are available through extensions such as dplyr [24] and Pandas [12]. Instead of re-reading CSV files, binary data files are created through ad-hoc serialization and large collections thereof are manually managed in complex folder hierarchies. Custom scripts are used for orchestration and changes without anything close to transactional guarantees. Data corruption and data loss is commonplace. No dedicated buffer management exists, files are typically loaded into memory in their entirety, often exceeding available resources and making yet another level of workarounds necessary. In essence, this zoo of one-off solutions, when viewed in their entirety from a distance, resemble a database management system that compares very poorly with state-of-the art solutions.

An orthogonal but related problem is found in the world of edge computing. Currently, edge nodes typically forward data to a central location for analysis. This is problematic due to bandwidth limitations especially on radio interfaces, and also raises privacy concerns. Performing analysis or pre-aggregation directly inside the edge node can help to limit the amount of data that has to be transferred to a central location, alleviating these problems.

To solve these problems, a *new class of data management systems* is required that efficiently supports embedded analytical use cases. These are systems that locally provide storage and querying capabilities together with transactional guarantees on large amounts of data. At the same time, tight integration with analysis tools is required to facilitate easy access to the data stored within the RDBMS and to eliminate traditional data transfer bottlenecks.

It might seem that it would only take minor re-engineering of existing systems to meet those requirements. Indeed we have tried to do just that in our previous work with MonetDBLite, which was derived from the MonetDB system [22]. MonetDBLite proved successfully that there is a real interest in embedded analytics. However, it also uncovered several issues that proved very complex to address in a non-purpose-built system. Those issues can be generalized to the general group of OLAP systems and originate from design assumptions that no longer hold in the embedded analytics scenario. For example, a big assumption in standalone OLAP systems is that they are running on hardware specifically dedicated to them that has no other tasks to complete otherwise. In the embedded scenario, this is not the case. A system administrator is expected to carefully monitor system health, and the reliability of hardware is typically higher, for example through the use of error-correcting memory and redundant disks. The following requirements for embedded analytical databases were identified:

1. **Combined OLAP & ETL Workloads.** High efficiency for OLAP workloads is required together with efficient support for bulk appends and bulk updates. Bulk appends occur when new data arrives. Bulk updates are common in "data wrangling" workloads to address missing or encoded values, unit conversions etc [5]. Concurrent data modification is common in dashboard-scenarios where multiple threads update the data using ETL queries while other threads run the OLAP queries that drive visualizations.

2. **Transfer Efficiency.** Efficient transfer of tables to and from the database is essential since not all tasks (e.g. training, classification or plotting) can be conveniently performed inside the database [10]. Since both database and application run in the same process and thus address space, there is a unique opportunity for efficient data sharing which needs to be exploited to allow for seamless passing of data back and forth between the application and the RDBMS.

3. **Resilience.** Consumer-grade hardware is more prone to hardware problems than server-grade hardware monitored by administrators. An embedded database needs to be able to detect these problems and prevent them from causing silent data corruption.

4. **Cooperation.** The system needs to gracefully adapt to resource contention. As the embedded database is no longer the sole inhabitant of the machine, it can no longer make constant use of all the underlying hardware as that would cause the underlying application to be starved for resources.

In this paper, we explore the requirements of embedded OLAP systems in detail and summarize the challenges that arise from these requirements. Afterwards, we discuss our attempts at tackling these challenges in DuckDB; our purpose-built embeddable relational database management system.

2. COMBINED OLAP & ETL WORKLOAD

The OLAP workloads encountered by embedded analytical systems are similar to workloads encountered by "standard" OLAP systems. The queries typically consist of large table scans and involve multiple aggregates and complex join graphs. The workloads also typically only target a subset of the columns of a large table. Much like in data warehouse workloads, bulk appends are also common place.

The extract-transform-load (ETL) process is also critical in analysis workflows. In fact, most analysis time is spent on data pre-processing [2]. Traditionally, the ETL process would be a separate pre-processing step before data is loaded into the data warehouse. However, in the embedded analytics case integrating both ETL and querying into the same system is feasible and desirable. Feasible because the database can directly scan existing files (e.g. CSV), reshape the result and then append it to a persistent table. Desirable because out-of-core processing, parallelization and transactional behaviour is also highly relevant in the ETL process.

ETL queries involve large batch updates or deletes. For example, a common data representation is to implicitly encode missing values as a special member of the domain, e.g. the value -999 [13]. The correct value can be reconstructed using the following query: `UPDATE t SET d = NULL WHERE d = -999`. In case there are many missing values, this query will update a large part of the column. As such, these queries are very different from traditional OLTP updates which touch only a minute fraction of the database.

There are several architectural implications stemming from these requirements. For the *query processor*, only a comparably low amount of CPU cycles per value can be spent. Vectorized or Just-in-time compilation query processing engines are the two state-of-the-art possibilities here [6]. Efficient bulk updates also require bulk granularity in the concurrency control. An important consideration here that in this scenario updates will typically affect only a small subset of columns, where deletes will affect entire rows.

The column-focused updates also influence the *on-disk storage format*. When some columns in a table are changed, the unchanged columns should not be rewritten in any way for performance reasons. Partitioning columns is still required though, otherwise changes again force an unnecessary rewrite of large amounts of data.

3. RESILIENCE

Ensuring transactional guarantees even in the presence of hardware failures is part of most DBMS implementations. Systems contain defensive code that will guarantee that committed transactions are persistent even if power fails immediately after the commit. In distributed installations, care is also taken to tolerate the intermittent loss of network connections. Redundant hard disk setups (RAID) are also recommended to tolerate loss of hard disks. Database server hardware will make use of error-correcting code (ECC) memory, which decreases the likelihood of bits being accidentally flipped in memory [4]. All of this occurs under the watchful eye of system administrators which are able to monitor and react to failures.

The hardware environment for embedded analytics data management is fundamentally different. The DBMS runs on user hardware, without RAID, ECC or system administrators. Hardware failure is more common, and the system administrator is the end user himself, who is unlikely to keep constant watch to ensure correctly functioning hardware. Providing transactional guarantees in this environment requires a radical departure of system building conventions. The hardware environment needs to be *distrusted* in every aspect to achieve a reasonable degree of resilience.

Failure	Pr[1st failure]	Pr[2nd fail | 1 fail]
CPU (MCE)	1 in 190	1 in 2.9
DRAM bit flip	1 in 1700	1 in 12
Disk failure	1 in 270	1 in 3.5

Table 1: 30-day OS crash probability [19]

Failure Rates. A recent comprehensive real-world study of desktop computers has found that for desktop computers running for 30 days, 1 in 190 will have a CPU failure, 1 in 1700 will have a bit flip in kernel memory, and 1 in 270 will have a disk failure [19]. Table 1 reproduces the numbers from this study. Notably, for systems that have suffered hardware failures previously the probability for the next hardware failure is increased by two orders of magnitude. Hence a system that has failed once is very likely to fail again.

Failure Types. There are two failure types for hardware: (1) detected errors and (2) silent errors. In case of a detected error, the hardware malfunctions but the hardware itself has detected that it has malfunctioned. Silent errors occur when the hardware malfunctions but continuous to operate as if nothing went wrong. In case of a detected error, we can simply report the hardware malfunction to the user and stop operation. Silent errors are much more dangerous, as they have the potential to cause silent data corruption. Rather than allowing data corruption through silent errors an embedded analytics DBMS needs to detect these errors and correct them if possible or cease operation entirely.

RAM Failure. Silent bit flips in RAM are a serious danger to DBMS integrity. An obvious approach to test its correct operation is to write a known pattern into RAM and read it back. This is not enough, however, because intermittent and data-dependent errors are missed [23]. Intermittent failures stem from interactions between adjacent cells in the memory chip. For example, writing to a cell might flip a neighboring cell. While this could be exhaustively tested in theory, exact knowledge of the chip layout would be required. If bit flips occur in the data to be written to the WAL or during checkpointing, database integrity is compromised. If bit flips occur in intermediate results or hash tables during querying, the DBMS might crash or return incorrect results. The failure modes are also interconnected; for example if a query result is written back to storage, a wrong query result will also compromise the persistent data's integrity.

Silent memory failures are extremely dangerous if an oblivious DBMS runs for an extended amount of time, slowly increasing the degree of data corruption. If finally detected at all, the stored data is likely useless. The issue is exacerbated in high-performance OLAP engines because they rely on larger chunks of memory to work correctly. Hence for an embedded analytical system, detecting these silent memory failures is crucial to ensure correct operation.

Persistent Storage Failure. The state of the art for hard disks is complex: The physical disk hardware will typically checksum individual blocks as they are written and check them as blocks are read again. In addition, error correcting codes are used to recover the original data transparently if an error is detected. If the data cannot be recovered, an exception is thrown which will be communicated to the kernel and the DBMS. Silent errors do occur however [16]. Even if a RAID setup exists, only some RAID levels (2 and up) store parity bits, and these are only used for the (expensive) data reconstruction process when a disk reports errors or stops responding. They are not used to detect silent errors.

There exist higher-level defenses to detect silent disk errors: some file system implementations such as ZFS will checksum any block written to disk and verify the checksum on read [25]. However, an embedded system cannot rely on running on a specific file system. Instead, the database itself needs to protect against these types of data corruption.

CPU Failure. Another hardware component that could fail is the *CPU*. CPUs contain self-checking, for example checking register and cache parity bits as well as enforcing several internal invariants [7]. If any discrepancy is detected, a machine-check exception (MCE) is issued, which stops the system from continuing. Hence, these issues are less critical because they cannot lead to silent data corruption [19].

State–of–the–Art. Standard disk failures and power outages are very well studied in database literature, and any ACID-compliant database handles these types of failures correctly through the use of e.g. Write-Ahead-Logs or rollback journals. Some DBMS implementations such as SQL Server also contain an optional feature to checksum and verify database pages on disk [14].

Unfortunately hardware errors beyond these issues are not very well studied. To the authors' best knowledge, the only work done on mitigating memory errors in the context of main memory systems is the work by Kolditz et al. [9, 8]. In this work error detection is efficiently implemented through the use of AN codes, resulting in resilience against random bit flips in the data while operating between 1.1× and 1.6× slower. While an excellent first step, more work needs to be done w.r.t. different data types, compression and adaptivity of the error detection. Another area that is unexplored in this work is detecting whether the hardware itself is broken and using that information. This detection can be performed similar to how memtest operates [23]. As bit flips are unlikely to happen on correctly functioning hardware, we could afford to use more lightweight error detection routines if we can verify that the hardware is working as expected. Another potential direction here is that often only specific areas of the RAM are broken whereas others function correctly [23]. A potential mitigation could therefore be figuring out which areas are broken and avoiding the use of those memory areas.

4. COOPERATION

Another unique property of embedded analytics is the co-inhabitation of DBMS and analysis application code on the same physical computer. Traditional OLAP systems rely on the Client-Server architecture and expect to be the sole application running on a dedicated database server. Hence, they typically assume all available computing resources are available and dedicated to the DBMS operations. For example, it is typical for a traditional OLAP DBMS to probe the amount of available CPU cores and main memory on startup and then configure internal buffers, parallelism etc. to use all the available resources. In an embedded analytics scenario, this approach is very problematic. The analysis application (for example a dashboard) will also need some hardware resources to operate properly; if all memory is taken by the DBMS, overall end-to-end system performance will suffer.

The conventional solution to resource contention between application and DBMS is to allow the system administrator to configure the memory limits that the DBMS is allowed to use for buffers etc. The task of deciding the limits however is non-trivial, how should for example the memory requirements of application and DBMS be weighed such that the overall system achieves best end-to-end performance? There exist statistical learning techniques to optimize DBMS configuration settings to maximize performance in isolation [20]. However, they do not consider the added complexity of resource sharing between application and DBMS. In addition, these approaches are reactive to the current DBMS workload but do not allow for adaptive resource sharing. It is likely that a front-end application will exhibit "bursty" CPU usage, to which the DBMS could react as to not interfere.

There are plenty of run-time choices in a DBMS that influence the resource consumption across the different hardware devices. An embedded OLAP system can monitor resource usage of all other running applications and then tweak its run-time behavior accordingly, such that the DBMS will use the resources that are under-utilized at the moment. This will lead to an faster end-to-end system response by not competing with the application in its time of need. In the following, we list some examples where resources can be traded off at run-time. The DBMS can make more impactful choices at this point than the operating system. Scheduling multiple competing applications is normally left to the OS, but because the algorithmic choices of the competing processes cannot be influenced by the OS, which is limited to swapping virtual memory in this case.

5. TRANSFER EFFICIENCY

Result set transfer from DBMS to external tools is notoriously slow [21]. The reasons for this are two-fold: Result set serialization to byte streams and value-based APIs. Serialization traditionally occurs due to the need to transfer a result set to a client program over a network connection. Network connections are byte streams, but result sets are two-dimensional structures of possibly complex values. While not particularly complex, the serialization and deserialization protocols used by virtually all popular DBMSs are not optimal and even if they were, data transfer over a network socket to another computer is limited by the available bandwidth, e.g. 1 Gbit/s.

A second bottleneck is present in the common value-based APIs to fetch data from query results. Common examples are the ODBC and JDBC APIs, but also the SQLite APIs. These APIs are convenient for programmers that for example seek to display a table with the query results. However, when transferring large result sets, the function call overhead for each value becomes excessive. In these cases, bulk access to e.g. a subset of a column in the query result is preferable.

A unique opportunity to avoid both issues exist in the embedded analytics system architecture. Because both DBMS and analytics tool are located in a single process' address space, data transfer can be particularly efficient. In fact, one of the motivations to develop embedded analytical DBMSs is to exploit this opportunity.

Improved transfer efficiency can potentially lead to a change in database workloads. In traditional client-server based database systems it is infeasible to transport large amounts of data outside of the RDBMS, requiring the user to write large and complex queries that perform many operations simultaneously. A highly efficient, or even zero-cost[22], data export allows the user to instead use multiple simple queries interleaved with application code to achieve the same result.

6. THE DUCKDB SYSTEM

We have started to develop the first representative of the new class of embedded OLAP DBMSs, *DuckDB*. DuckDB is a new purpose-built embeddable relational database management system. DuckDB is available as Open-Source software under the permissive MIT license[1].

OLAP & ETL: DuckDB uses a vectorized interpreted *execution engine* [1] that is optimized for OLAP queries. This approach was chosen over Just-in-Time compilation (JIT) of SQL queries [17] for portability reasons. JIT engines depend on massive compiler libraries (e.g. LLVM) with additional transitive dependencies. For practical embeddability and portability, external dependencies have been found to be extremely problematic.

The execution engine executes the query in a so-called *"Vector Volcano"* model. Query execution commences by pulling the first "chunk" of data from the root node of the physical plan. A chunk is a horizontal subset of a result set, query intermediate or base table. The chunk consists of a set of column slices. This node will recursively pull chunks from child nodes, eventually arriving at a scan operator which produces chunks by reading from the persistent tables. This continues until the chunk arriving at the root is empty, at which point the query is completed.

DuckDB provides ACID-compliance through Multi-Version Concurrency Control (MVCC). Because queries might run for considerable time, lock-free Multi-Version-Concurrency control is a sensible choice that is able to provide multiple consistent views on the same dataset without blocking query progress[18]. We implement HyPer's serializable variant of MVCC that is tailored specifically for hybrid OLAP/OLTP systems [18]. This variant updates data in-place immediately, and keeps previous states stored in a separate undo buffer for concurrent transactions and aborts.

DuckDB uses a single-file storage format to store data on disk. The file format is designed to support efficient scans and bulk updates, appends and deletes. While single-value or single-row updates are supported, their efficiency is not a design goal. The format allows to scan individual columns and skip irrelevant blocks of rows during a scan.

The fact that the database only consists of a single file was inspired by SQLite [3] and repeated user requests for this feature in MonetDBLite. As an exception, the WAL is written to a separate file until consumed by a checkpoint. The storage file is partitioned into fixed-size blocks of 256KB which are read and written in their entirety. The first block contains a header that points to the table catalog and a list of free blocks. The catalog contains pointers to lists of schemas, tables and views. Each table consists of a number of blocks that hold the data. Checkpoints will first write new blocks that contain the updated data to the file and as a last step update the root pointer and the free list in the header atomically.

Resilience: DuckDB computes and stores check sums of all blocks in persistent storage and verifies this as blocks are read. This protects against bit flips in the persistent storage which would go unnoticed or cause inconsistencies.

As next steps, we plan to implement protection by testing whether regions in memory are broken or not. There exist approximate memory error detection algorithms like "moving inversions" that can uncover memory issues in a generic way [23]. However, these tests create significant traffic on the memory bus, it is thus not feasible to constantly test the entire memory. As a compromise, we plan to integrate memory tests into the buffer manager, which will test all buffers on allocation to detect existing errors and periodically to detect new errors.

Cooperation: DuckDB for now allows the user to manually set hard limits on memory and CPU core utilization. In the future, we envision an adaptive resource usage scheme where all levels of the system react to an observed resource utilization scenario in real time. For example, RAM and

[1]https://www.duckdb.org

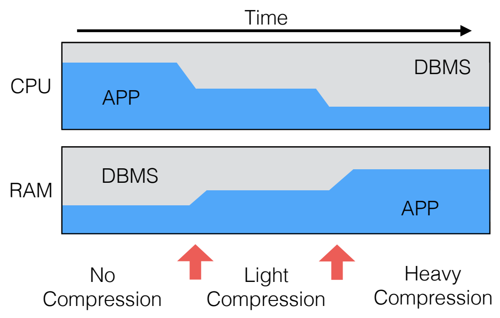

Figure 1: Example reactive resource usage pattern

CPU can be traded-off against each other rather elegantly. On the physical plan level, a hash join can be transparently replaced with a out-of-core merge join. The hash join uses a large amount of main memory to store the hash table, but few CPU cycles to compute the actual join result because of its lower complexity class. The merge requires fewer main memory resources to run, but $\mathcal{O}(n \log n)$ CPU cycles as well as disk IO. If the DBMS detects that the application currently uses a large amount of main memory but not a lot of CPU cores, it can switch to merge join to reduce the load on RAM and use CPU cores and the disk instead.

On the engine level, we can also choose to compress temporary structures like hash tables in memory with different compression algorithm. Figure 1 shows an example of how a reactive intermediate compression might operate. As the RAM usage of the application increases, the DMBS chooses first lightweight compression to reduce its memory footprint at the expense of extra CPU cycles. As the RAM usage of application increases further, the DBMS switches to a heavy compression algorithm that will further reduce the memory footprint. The compression and associated overhead will reduce the response time of the DBMS, however, the end-to-end response time is improved by reducing memory pressure for the application.

Transfer Efficiency: DuckDB implements a highly efficient client API. The API allows the client application to essentially become the root operator in the physical query processing plan. The application will poll the database engine for chunks of data to be computed. Once a chunk has been filled, it is handed over to the client application. This continues until the query has finished executing. The chunks are represented as a collection of column slices, exactly identical to the internal representation. Because DuckDB is an embedded DBMS, the database and the client application share the same address space, the chunk is handed over without requiring copying. The same is true for appending data to tables, the client application can fill chunks with its data. Once filled, they are handed over to DuckDB and appended to persistent storage. All APIs are built around bulk value handling to prevent function call overhead from becoming a bottleneck. The bulk API allows efficient data transfer to and from R and Pandas/NumPy.

7. CONCLUSION

Traditional DBMS implementations have been largely ignored by Data Science practitioners because the research community failed to see and support their use case. In order to recover this lost terrain, a paradigm shift is required in thinking about what constitutes a "proper" DBMS. In this paper, we have argued for the focus to shift on embedded analytical data management.

Research Challenges. We have described the pressing issues as well as our first steps in addressing them in the DuckDB system. However, many open questions remain:

- How can we efficiently handle bulk updates and deletes in ETL-like workflows?
- How does a modern data analysis workload look like beyond TPC-DS? What part of ETL needs to be part of benchmarks?
- How does fast data transfer impact query workloads? Are large complex queries still required when data export is nearly free?
- How can we increase DBMS reliability in light of silent hardware errors without compromising performance?
- How can we implement effective resource sharing through reactive DBMS reconfiguration?

8. REFERENCES

[1] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, pages 225–237, 2005.

[2] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[3] R. Hipp. Database file format. https://www.sqlite.org/fileformat.html, 2019.

[4] Intel. Delivering Resilient and Reliable Workstations: The Role of ECC Memory. 2015.

[5] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.

[6] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything You Always Wanted to Know About Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.*, 11(13):2209–2222, Sept. 2018.

[7] A. Kleen. Machine check handling on Linux. 2004.

[8] T. Kolditz, D. Habich, W. Lehner, M. Werner, and S. T. de Bruijn. AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection During Database Query Processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1619–1634, New York, NY, USA, 2018. ACM.

[9] T. Kolditz, T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Online bit flip detection for in-memory b-trees on unreliable hardware. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 5:1–5:9, New York, NY, USA, 2014. ACM.

[10] J. Lajus and H. Mühleisen. Efficient data management and statistics with zero-copy integration. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 12:1–12:10, New York, NY, USA, 2014. ACM.

[11] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 294–305, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[12] W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[13] Q. McMullen. How to Represent Missing Data: Special Missing Values vs. 999999999. 2001.

[14] Microsoft. *Microsoft SQL documentation*, 2017.

[15] P. Mooney. Kaggle Machine Learning & Data Science Survey. Oct. 2018.

[16] J. Myers. Data Integrity in Solid State Drives: What Supernovas Mean to You. Feb. 2014.

[17] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[18] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689, 2015.

[19] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[20] A. Pavlo, G. Angulo, J. Arulraj, et al. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017.

[21] M. Raasveldt and H. Mühleisen. Don't hold my data hostage - A case for client protocol redesign. *PVLDB*, 10(10):1022–1033, 2017.

[22] M. Raasveldt and H. Mühleisen. Monetdblite: An embedded analytical database. *CoRR*, abs/1805.08520, 2018.

[23] P. Software. MemTest-86 User Manual. Technical report, 2018.

[24] H. Wickham, R. François, L. Henry, and K. Müller. *dplyr: A Grammar of Data Manipulation*, 2018. R package version 0.7.8.

[25] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 29–42, 2010.