SwingDB: An Embedded In-memory DBMS Enabling Instant Snapshot Sharing

Qingzhong Meng[1], Xuan Zhou[1], Shiping Chen[2], and Shan Wang[1]

[1]MOE Key Laboratory of DEKE, Renmin University of China, Beijing 100872

[2]CSIRO Data61, PO Box 76, Epping, NSW 1017, Australia

Abstract. Data transmission between an in-memory DBMS and a data analytical program is usually slow, partially due to the inadequate IPC support of modern operating systems. In this paper, we present SWING, a novel inter-process data sharing mechanism of OS, which allows processes to share physical memory through an instant system call. Based on SWING, we develop an embedded in-memory DBMS called SwingDB, which enables data analytical applications to access databases in their own memory space, instead of resorting to traditional inter-process communication. Extensive experiments were conducted to demonstrate the advantage of such a DBMS-OS co-design.

1 Introduction

As the capacity of RAM keeps growing exponentially, it has become an important instrument for big data processing. In the emerging paradigm of in-memory computing, people prefer to store an entire database in RAM, and perform data processing completely on RAM. This can substantially speed up the processes of data manipulation and data analysis. However, most data analytical workflows are multi-stage. They usually involves a number of data processing programs and services that cooperate to generate results. In a typical case of data analysis, data is usually stored in a DBMS; when an analytical process starts, it first issues queries (e.g., in SQL) to the DBMS to retrieve required data; then, the data is passed to a data analytical program to perform data preparation and statistical analysis; finally, a data visualization program is used to present the analytical results to the end user. Sometimes, large volume of data needs to be transmitted across various programs and services. If data transmission is slow, as it always is, it will obliterate the performance advantage of in-memory computing.

To the best of our knowledge, the mechanisms of inter-program data exchange provided by today's operating systems can hardly meet the performance requirement of in-memory computing. The IPC Mechanisms of FIFO and Socket appear extremely slow, as they need to move data physically. While the shared memory mechanism does not move data, the programs using shared memory have to deal with space allocation and data synchronization on their own, which incurs extra cost. On the one hand, when multiple programs are tied to a single piece of

shared memory, they become tightly coupled, which may raise the cost of software development and maintenance. On the other hand, it is sometimes unsafe to allow applications to access the memory space of a DBMS – once an application is malfunctioning, it may impair the integrity of the data.

In this paper, we introduce a new copy-on-write solution to inter-process data sharing. It is fast and convenient. In contrast to shared memory, it enables loose coupling between data processing programs, so that it matches today's practice of software engineering. We call our approach *SWING*, which analogizes the transmission of data to how Tarzan swings from a tree to another. The memory allocated by the SWING mechanism is called *COW Memory*, which is a type of *virtual memory* with the following characteristics:

1. Two chunks of COW memory can be mapped to the same set of physical memory pages, to share data.
2. Modifications on different chunks of COW memory (that share physical memory pages) are isolated through copy-on-write.

When a process wants to share data to another process, it can place the data in a COW memory area and let the other process allocate another COW memory area that is mapped to the same physical memory space. After the allocation, both processes can see the same contents in their own COW memory areas. This approach allows us to avoid physical movement of data. Afterwards, the two processes can modify their own COW memory areas independently. A copy-on-write mechanism makes sure that their modification should be invisible to each other, so that no synchronization is required.

Based on SWING, we create a new in-memory DBMS called SwingDB. SwingDB works as an embedded DBMS, such that each application operates on a database in its own memory space, without incurring inter-process communication. Each database instance of SwingDB is completely placed in a COW memory area, so that independent applications can share the snapshots of their databases using the SWING mechanism. SwingDB is especially suitable to multi-stage in-memory data processing, in which several loosely coupled programs cooperate in performing data analysis.

We implemented the SWING mechanism in Linux[1], and then constructed our SwingDB system by re-engineering an open-source in-memory DBMS called SuperSonic. We conducted experiments to characterize the performance of SWING and SwingDB. We also compared SwingDB against traditional in-memory DBMS, to demonstrate its suitability in in-memory data analytics.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 presents the design and implementation of SWING. Section 4 introduces SwingDB and discusses its potential applications. Section 5 presents the results of our experimental evaluation. Section 6 concludes the paper and discusses our future research plans.

[1] Source code: http://swinglinux.github.io/swing/

2 Related Work

In-Memory Databases

Data movement is an expensive operation for data intensive applications, while it often occurs between a DBMS and a data analytical application. Especially when conducting large scale statistical analysis, we need to transmit large volumes of data from a DBMS to an analytical tool, such as R or SAS. In the context of in-memory computing, such data transmission is heavy and may kill the performance.

Folk wisdom believes that it is cheaper to move programs to data than to move data between programs. To this end, a number of database systems integrate components of data analysis and data mining [5], and expect applications to perform data analysis within the database systems. Some in-memory DBMS even combines the database server and the application server into one single system, to minimize the cost of data movement [7, 15]. However, the tight coupling between database systems and data analytical tools is a double-edged sword – while it reduces the communication cost, it raises the cost of software engineering [4, 8], which regards "separation of concerns" as an essential principle. In many cases, developers of a DBMS do not know what analytical algorithms applications will demand, while developers of applications have little knowledge about how a DBMS works. The design of SwingDB aims to minimize the cost of data exchange, while keeping the coupling between DBMS and data analytical programs as loose as possible.

The most relevant work to SwingDB is the in-memory DBMS named Hyper [9]. Hyper was designed to support OLTP and OLAP simultaneously. Hyper's main process is responsible for maintaining the integrity of a database and performing updates. When an analytical request arrives, the main process invokes the system call $fork()$ to initiate a child process. As the child process shares the memory space of the main process, it can immediately see a complete snapshot of the in-memory database and conduct data analysis independently on that snapshot. A copy-on-write mechanism is employed by the operating system to guard the isolation between the parent and child processes. To the best of our knowledge, SAP HANA [7, 15] also utilizes $fork()$ to share data between the DBMS and the analytical applications. Although the $fork()$ approach successfully avoids the cost of data movement, it does not offer good usability. To perform data analysis, an application needs to pack its analytical program and ships it to the DBMS (e.g., in the form of dynamic link libraries). This makes the development of analytical applications complicated. On the other hand, it does not allow joint analytics over multiple database systems, as multiple parent processes cannot share the same child process. By comparison, SwingDB is more flexible and efficient than the $fork()$ approach.

There have been several recently proposed techniques which utilizes copy-on-write to realize concurrency control on in-memory data [11, 14, 2]. As their use cases are different from that of SWING, we regard them less relevant to our work.

Inter-Process Communication

The most commonly used inter-process communication methods include named pipe (FIFO), socket and shared memory. When transmitting data, both FIFO and socket need to move data physically. In particular, they need to copy data from the source to a buffer, and then copy it from the buffer to the destination. Sometimes multiple layer buffers exist, such that data needs to be copied for multiple times [12]. Copying data physically is expensive in the context of in-memory computing.

Shared memory is so far regarded as the fastest IPC approach, as it does not move data physically. Normally, an operating system offers two modes to access shared memory. In the *shared* mode, a process' writes to the shared memory are completely visible to the others. In the *private* mode, a process' writes to the shared memory are only visible to itself – once it attempts to write to the shared memory, copy-on-write operations will be invoked to hide the write from the other processes.

Applying shared memory to data transmission between a DBMS and an analytical process, the DBMS can choose the shared mode, and the analytical process can choose the private mode. Although this setting seems safe, it is not as powerful as our SWING approach. First, the DBMS' writes on the shared memory are constantly visible to the analytical process. As a result, concurrency control is required, to prevent the DBMS from further updating the snapshot. In SWING, after the transmission of data, the DBMS' updates are no longer visible to the receiver. Thus, synchronization is not required. Second, the analytical process cannot further share the memory to other processes. Sometimes, an analytical workflow is multi-stage. For instance, in the early stage of data analysis, some intermediate results are generated and added to the data as annotation; then, the annotated data is passed to the subsequent stage for more advanced analysis. Such mult-stage data processing is difficult to realize on shared memory. In contrast, SWING allows recursive data sharing, which suits mult-stage data processing much better.

3 The SWING Mechanism

3.1 The Data Sharing Model

The data sharing model of SWING is illustrated in Figure 1. Processes A and B are two applications. They both send copies of their data to Process C, which performs data integration and data preparation. After the work, Process C sends its copies of data, with its modification, to the Processes D and E, which are responsible for data analysis. Such a data transmission process can be repeated infinitely. After the data sharing, all processes can work on their own copies of data independently, such that their modification of the data is invisible from each other.

The same effect can be achieved by FIFO and socket. In contrast to those approaches, the SWING method does not replicate or move data physically. It

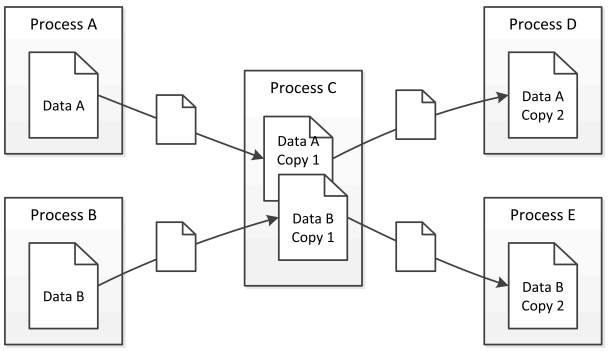


Fig. 1. Data Sharing Model of SWING

just maps the physical memory space containing the data to the virtual memory space of the target processes. The actual replication is delayed to the time when a process attempts to modify a block of the data – upon modification, a copy-on-write operation is invoked and a new version of the data block is created. In typical data processing scenarios, write operations are much less frequent than read operations. Therefore, the overhead of data replication can be minimized through SWING.

Using the SWING mechanism, the processes involved in data sharing are kept loosely coupled – they share only data and the code / library for interpreting data. They do not share any controlling code, such as the code to ensure data consistency. According to the principles of software engineering [3], data coupling is much more flexible than code coupling. If using shared memory, the processes have to share the code dealing with concurrency control.

Such a data sharing model cannot be realized by the $fork()$ approach either, as Process A and Process B cannot both be the parent of Process C.

3.2 The Interfaces

We implemented five system calls to realize the SWING mechanism. In SWING, the memory space used to transmit data is called COW (Copy-on-Write) memory.

1. *long createarea*(*long length*) A process uses this system call to apply for a COW memory area. The input parameter *length* indicates the size of the COW memory. The return value (64 bits on an x86_64 platform) contains two parts of information. The lowest 12 bits are a *token*, a unique identifier in the whole system to identify the COW memory area. It works as a file descriptor of shared memory. The other 52 bits point to the start address of the COW memory. As the size of a memory page is normally 4KB, the address of a COW memory area is aligned with 4KB.

2. *long hook*(*int token*) A process uses this system call to obtain a new COW memory area and maps it to the physical memory space of an existing COW memory area. Its input parameter is *token*, representing an existing COW memory area. The returned value of *hook*() is the same as that of *createarea*(), which contains a token and an address. The returned *token* is different from the input *token*, as they represent different COW memory areas.

3. *void enablehook*(*int token*) A process uses this system call to inform the operating system kernel that a COW memory area is ready to be mounted (by calling *hook*()). This system call is used to ensure data consistency. Before a process finishes modifying the data in its COW memory area, it may not wish other processes to hook the area and see a dirty version of the data. If a COW memory cannot be mounted, *hook*() returns -1.

4. *void disablehook*(*int token*) A process uses this system call to inform the operating system kernel that a COW memory area cannot be hooked. It reverses the effect of *enablehook*(). When a COW memory area is just created, it is automatically disabled.

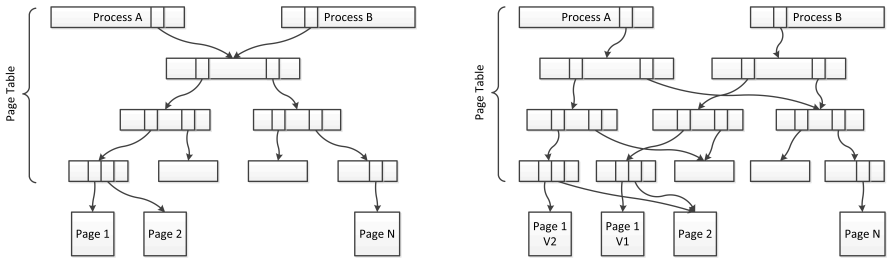
5. *void release*(*int token*) A process uses this system call to release a COW memory area created by *createarea*() or *hook*(). After the call, the input *token* is released and available to represent a new COW memory area. When a process terminates, its COW memory areas will be automatically released.

3.3 The Implementation

In modern operating systems, when a process accesses a byte in its virtual memory space, the *virtual address* of the byte will first be translated to a *physical address*, through which the processor addresses the physical memory on the memory bus [1]. In a typical x86-64 architecture, a 4-level page table is used to perform the translation. Basically, a linear address space is divided into pages, normally 4KB in size. Each virtual memory page of a process is mapped to a physical page in the physical memory, and the mapping is recorded in the page table of the process.

Page entries of different page tables can be mapped to the same physical page, such that multiple processes can share the same segment of physical memory. For instance, when $fork()$ is invoked, Linux will replicate the entire page table of the parent process to that of the child process, so that their memory space are identical. After $fork()$, all the page entries of the page tables are marked as *read-only*. When a process attempts to write to a page marked as read-only, a page fault occurs; then the operating system will allocate a new physical memory page to the process, copy the contents of the original page to the new page and flags the new page as *writable*. Then, the write operation can be conducted on the new page. This is known as a typical copy-on-write process.

One possible way to implement SWING is to reuse the mechanism of $fork()$. Instead of replicating an entire page table as what $fork()$ does, we can replicate



(a) after *hook*() (b) after Process A updates Page 1

Fig. 2. Memory Sharing on SWING

only the fraction of the page table that corresponds to the COW memory area. However, partial replication of page table is still costly, especially when the COW memory area is big. In addition, as the replication procedure will block both the sending and receiving processes, it may prevent *hook*() from being frequently invoked. To avoid the cost of replication, SWING abandons the $fork()$ approach and allows different processes to share page tables or parts of page tables [13] as well. This is illustrated in Figure 2(a). When *hook*() is invoked, the subtree of the page table of Process A that corresponds to the COW memory area is entirely shared to Process B. The whole procedure only requires an update on a single entry in the page table of Process B. Afterwards, both processes can see the same contents in their COW memory areas.

After the *hook*() operation, when a process attempts to update a page in the shared subtree, a copy-on-write process is invoked and the shared subtree is split into a double rooted tree. This is illustrated in Figure 2(b). When Process A writes on Page 1, a new version of Page 1 is created to receive the write, and a new path to this version is instantiated and merged into the subtree. Afterwards, the subtree contains two roots, belonging to Processes A and B respectively. As updates continue to occur on other pages, the part of the subtree rooted at A and that rooted at B will become more and more detached. To ensure the correctness of the copy-on-write mechanism, we only need to mark the shared parts of the subtree as read-only and the parts exclusively belonging to one process as writable. When one of the processes quits or releases its COW memory, only the part of the subtree exclusively belonging to that process is removed.

In our SWING mechanism, the data sharing step (i.e., invocation of *hook*()) is extremely fast and almost nonblocking. Therefore, it can be invoked frequently. Although it incurs additional copy-on-write overhead during updates, this overhead is controllable.

4 SwingDB

SwingDB is an embedded in-memory DBMS dedicated for data analytical applications. It provides the basic functionality of a stand alone relational DBMS. In particular, it allows an application to include an entire database in its memory

space, so as to avoid expensive inter-program communication. SwingDB achieves this by utilizing the SWING mechanism.

4.1 The Functionality of SwingDB

SwingDB works as an embedded DBMS – an application requiring data management functionality can have the entire SwingDB system in its program; each database instance of SwingDB resides entirely in the memory space of the application's process, so that data manipulation does not require any inter-process communication. Another application attempting to perform data analysis can retrieve a snapshot of the database and places it in its own memory space, so that it can access the database cheaply too. Moreover, the second application can further share its snapshot to a third application for more advanced data analysis. This process is illustrated in Figure 3.

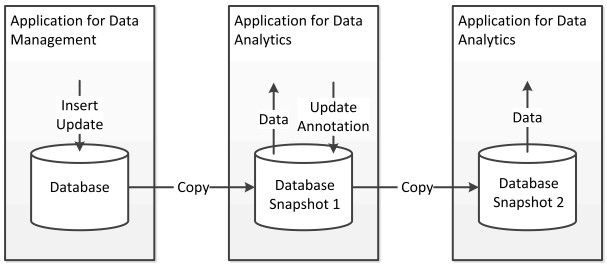


Fig. 3. SwingDB Allows In-Process Data Access

SwingDB assigns each snapshot of a database a unique name. An application can retrieve a database snapshot into its space simply through the following function call:

$$bool\ getsnapshot(string\ proposedname, string\ targetname)$$

Basically, this function retrieves a database snapshot named *targetname* into the memory space of the active process, and names the new snapshot as *proposedname*. Afterwards, the process can perform standard database operations on its own snapshot through SQL queries. Or it can directly address the data records in the snapshot through low-level interfaces, to perform advanced data analysis and data mining. As a result, each database snapshot of SwingDB resides in only one process and can be accessed only within that process. When an application no longer needs a database snapshot or regards a snapshot outdated, it can abandon the snapshot through the following function call:

$$bool\ discardsnapshot(string\ name)$$

The advantage of SwingDB lies in the efficiency of its snapshot sharing, which is almost costless. By utilizing the SWING mechanism, the snapshot sharing of SwingDB requires no physical data movement.

4.2 The Implementation of SwingDB

SwingDB stores each database instance or snapshot in a single COW memory area of SWING. When a database is created, a new COW memory area is allocated for the database. Initially, the COW memory area consumes no physical memory, though it takes 512GB of virtual space. When more data is inserted into the database, more physical memory is allocated to the corresponding COW memory area. The memory space manager of of SwingDB is responsible for managing the space allocation within the COW memory areas, as shown in Figure 4. Based on such a design, the function *getsnapshot*() can be realised through the SWING function *hook*(), which is highly efficient.

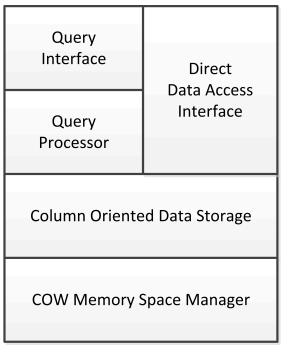


Fig. 4. The Basic Architecture of SwingDB

We built our SwingDB on top of Supersonic[2], an in-memory column store developed by google. We re-engineered the storage layer of Supersonic to move the entire storage space into COW memory of SWING. To enable snapshot sharing, we locate the meta data of a database in the first segment of its COW memory area, so that a database can be easily identified by a new process. The source code of SwingDB can be found in GitHub[3].

4.3 Application Scenarios

In a traditional setting of multi-stage data analysis, data is physically transmitted from program to program. Each program receives data from its previous programs, performs a certain type of data processing and sends its results, along with the original data, to the subsequent programs. Transmission of data can be conducted in several ways – each program can transmit data individually, or all programs can send and receive data to and from a mediated database. Such multi-stage data analysis is commonly used in modern scientific study [17, 6, 10].

[2] https://code.google.com/archive/p/supersonic/
[3] Source code: http://swinglinux.github.io/swing/

When performing in-memory data analysis, we store data entirely in RAM. In this context, we expect the whole process of data analysis can be finished in a few seconds, such that the data analytical application can become really interactive [18]. Such a speed cannot be achieved, if data has to be moved physically from program to program or between programs and a mediated database. SwingDB provides an efficient solution to multi-stage in-memory data analysis. Data is always stored in SwingDB, and the snapshots of the data are passed around by the programs. Each program retrieves the data snapshots from its previous programs, performs data processing within its own memory space and passes the resulting snapshots to its subsequent programs. No matter how complex the workflow is, no physical data movement is actually performed.

5 Performance Evaluation

We conducted experiments to study the performance characteristics of the SWING mechanism and SwingDB. The experiments were conducted on a HP Z820 workstation, equipped with two 2.60GHz Intel Xeon processors E5-2670 and 64GB DDR3 RAM. The operating system installed on the workstation was CentOS 7.1.

5.1 Overheads of SWING

Our first set of experiments was intended to measure the overhead of data transmission. We compared the SWING mechanism against FIFO (a.k.a. pipe) and shared memory. For FIFO, the overhead was measured by the execution time of the entire data transmission process. As to shared memory, we assume that the receiving process needs to block the sending process when reading the data. Its overhead is measured by the execution time of *mmap*() and the time for locking and scanning the data. Fine grained concurrency control can be used to improve the concurrency of shared memory. However, due to the complexity of the implementation of a fine grained concurrency controller, we did not consider it in our experiments. For SWING, its overhead is measured by the executing time of *hook*(). In the experiments, we varied the amount of data from 1GB to 8GB. The measured overheads are shown in Figure 5. As expected, SWING is way faster than FIFO and shared memory when transmitting data.

Our second set of experiments was intended to measure the overhead incurred by copy-on-write operations. In the experiments, we let Process A allocate a COW memory area of 8GB and keep updating the data in the area; then, we let Process B *hook* the COW memory periodically. Thus, Process A's updates will incur copy-on-write operations. Our experiments were intended to quantify how much Process A is slowed down by copy-on-write.

In the first experiment, we let Process A perform sequential update. We varied the frequency of the invocations of *hook*() and measured the variation of Process A's throughput. We compared the results against the case where

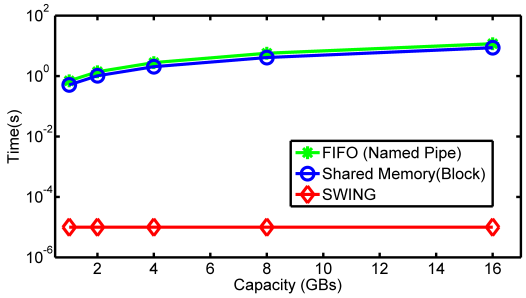
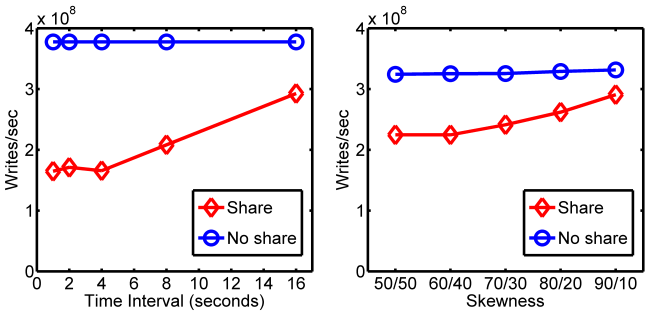


Fig. 5. Overhead of Data Transmission



(a) sensitivity to frequency (b) sensitivity to locality

Fig. 6. Overhead of Copy on Write

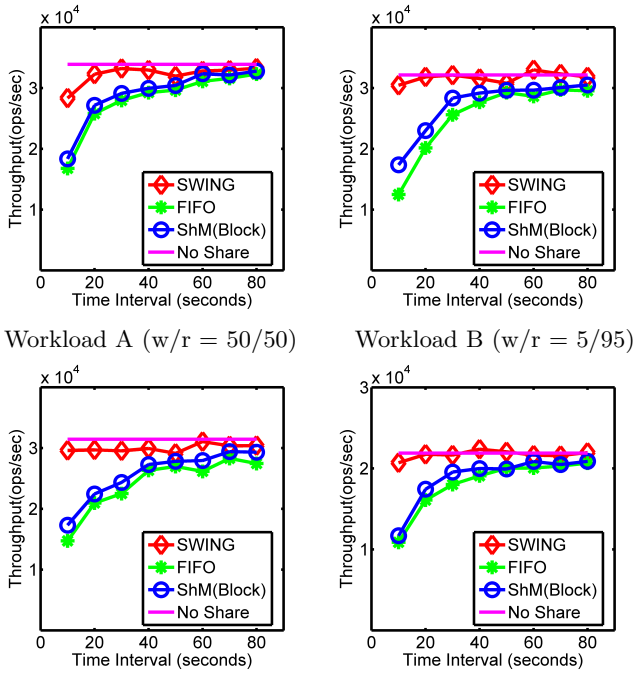
no data sharing was performed. As shown in Figure 6(a), copy-on-write operations do affect performance. The influence increases as we raise the frequency of data sharing. Nevertheless, the overhead is controllable. In the worst case, the performance of updating drops by around 50%. If we keep the frequency of data sharing to a moderate level (e.g. once per 20 seconds), the performance loss can be minimized.

In the second experiment, we fixed the frequency of *hook*() to once per 8 seconds and let Process A perform random updates. We varied the skewness of update distribution and measured the variation of Process A's throughput. This allows us to see how data locality affects the overhead of copy-on-write. As shown in Figure 6(b) (on x-axis, 80/20 means that 80% of updates were performed on 20% of data), when the locality of updates increases, the penalty caused by copy-on-write drops. In most real world applications, data accesses normally show strong locality. Thus, the overhead of copy-on-write should not be outstanding most of the time.

5.2 Experiments on OLTP Workload

Our second set of experiments aimed to evaluate how the copy-on-write operations of SWING affects the performance of database update. As SwingDB does not specialize in OLTP, we used Redis, an OLTP oriented in-memory database. We applied SWING to Redis, by moving the entire storage space of Redis to a single COW memory area. Then, we let a data analytical process to hook the COW memory of Redis periodically. After each *hook*() operation, the analytical process performs a sequential scan of the data. At the same time, we ran the YCSB Benchmark on Redis, to see how data sharing affects Redis' performance. We compared the COW mechanism against FIFO and shared memory. (When using shared memory, the analytical process needs to block Redis while reading the data, as Redis does not support fine grained access control.)

In the experiments, we set *recordcount* of YCSB to 900,000 for Workloads A,B,C,F and 500,000 for Workloads D,E. With such a data size, Redis' storage space can be accommodated in an 8GB COW memory area. We set *operationcount* of YCSB to 10,000,000. As to the other parameters, we used the default values of YCSB. We varied the frequency of invocation of *hook*() from once per 10 seconds to once per 80 seconds. The performance of Redis is shown in Figure 7. (Due to limited space, we only show the results on Workloads A,B,D,F.)



Workload A (w/r = 50/50) Workload B (w/r = 5/95)

Workload D (w/r = 5/95) Workload F (w/r = 50/50)

Fig. 7. Performance on YCSB

As we can see, SWING does not have significant influence on Redis's normal work. Even when the frequency of data sharing went up to once per 10 seconds, we still could not see any significant drop of Redis' performance. (The TPS of the original Redis is a bit more than 30 thousands. While some recent experimental in-memory systems [16] claim to achieve a million TPS, such throughput does not apply to Redis, which is a single threaded full-fledged system.) As updates in YCSB show strong locality, the performance penalty caused by copy-on-write seems quite limited. (Note that YCSB's data accesses follow the Zipfian distribution.) For update intensive workloads, such as Workloads A, the performance of Redis falls slightly when SWING is used. This indicates that copy-on-write can be an overhead for update intensive applications, though its influence is limited. In contrast to SWING, FIFO and shared memory did affect the performance of Redis significantly, especially when the frequency of data sharing is high.

5.3 Experiments on SwingDB

To evaluate the performance of SwingDB, we compared it against Vectorwise (Version 4.2.0), one of the most efficient in-memory DBMS specializing in OLAP. While it is difficult for SwingDB to beat Vectorwise in OLAP performance, SwingDB is way faster than Vectorwise in data transmission. We put both systems in a workflow of data analytics, consisting of a DBMS and a data analytical application. SwingDB and Vectorwise play the role of the DBMS, which is responsible for data management and query processing. The analytical application retrieves data from the DBMS through SQL queries, and performs advanced analytical study on the data. To work with Vectorwise, the application makes use of the APIs of Vectorwise to establish a connection with the database, issue queries and move the query results from Vectorwise to its own space. To work with SwingDB, the analytical application first retrieves the whole snapshot of the database into its own space, and then executes SQL queries and advanced analysis on the data. The main difference is that SwingDB does not require inter-process data movement.

In our experiments, we created a simple database composed of only one relational table and loaded 10GB of data into the database. To retrieve data, we used a selection query with varying selectivity. When data is retrieved, the application conducts statistical analysis over the data, which mainly consists of calculation of standard deviation. The execution time of the whole analytical process was recorded and plotted in Figure 8.

As we can see, when the size of query results is very small (i.e., selectivity as low as 1%), SwingDB does not necessarily perform as well as Vectorwise. In this case, query execution consumes the majority of the time, and SwingDB is not as optimized as Vectorwise in query processing. When the size of query results become larger (i.e., selectivity higher than 5%), SwingDB starts to outperform Vectorwise. In this case, the transmission of query results from Vectorwise to the application becomes more expensive than query execution. When large amount of data needs to be transmitted, the superiority of SwingDB becomes obvious.

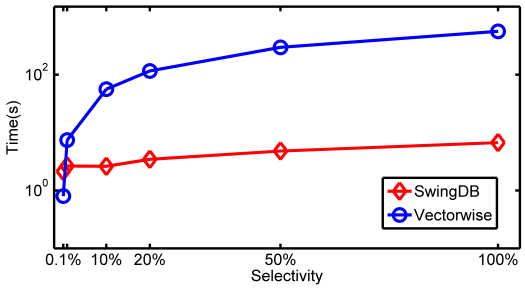


Fig. 8. Execution Time of Data Analysis

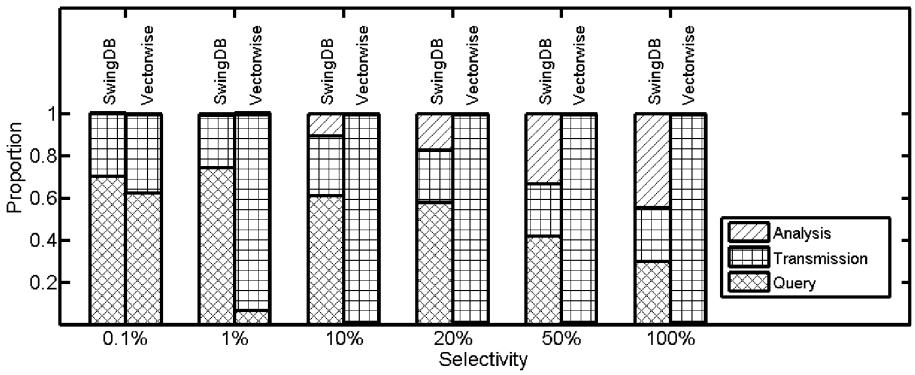


Fig. 9. Breakdown of the Execution Time

For instance, when the selectivity is as high as 100% and 2GB of data needs to be transmitted, SwingDB is faster than Vectrowise by two orders of magnitude.

If we break down the execution time of data analysis, as shown in Figure 9, we can see that the majority of the cost for Vectorwise was incurred by inter-process data transmission. In large scale data analysis, data transmission can be intensive. It may obliterate the performance advantage of in-memory databases. In contrast, SwingDB successfully avoids physical data movement by utilizing SWING. Therefore, it can be much more efficient in multi-stage data analysis than traditional in-memory DBMS.

6 Conclusion

In this paper, we introduced SWING, a new inter-process data sharing mechanism, and SwingDB, an embedded database system built on top of SWING. Different from traditional database systems, SwingDB is able to share database snapshots instantly among processes. This DBMS-OS co-design proves to be suit-

able for multi-stage data analytics, in which multiple loosely coupled systems or components cooperate to generate analytical results. (We believe that such multi-stage settings / cases will be increasingly common for future data analytics, as witnessed by today's scientific data management [17, 6].) We conducted extensible performance evaluation on our systems. The results showed that SwingDB is highly efficient in snapshot sharing and its extra overhead caused by copy-on-write operations is moderate and controllable.

As our future work, we will continue to enrich the functionality of SwingDB as both a database system and a tool for advanced data analytics. We would also like to invite the communities of DB and OS to join our effort of making SWING a standard instrument in data processing platforms.

References

1. Intel® 64 and ia-32 architectures software developer's manual volume 1: Basic architecture. *Intel Corporation* (August 2012).
2. Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. *Communications of the ACM 55*, 5 (2012), 111–119.
3. Beck, F., and Diehl, S. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 354–364.
4. Castellano, G. V. System object model (SOM) and Ada: an example of CORBA at work. *ACM Sigada Ada Letters XVI* (1996), 39–51.
5. Chaudhuri, S. Review - integrating mining with relational database systems: Alternatives and implications. *ACM SIGMOD Digital Review 2* (2000).
6. Curcin, V., and Ghanem, M. Scientific workflow systems - can one size fit all? In *2008 Cairo International Biomedical Engineering Conference* (Dec 2008), pp. 1–9.
7. Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. Sap hana database: data management for modern business applications. *ACM Sigmod Record 40*, 4 (2012), 45–51.
8. Garlan, D., Schmerl, B., Garlan, D., and Schmerl, B. Component-based software engineering in pervasive computing environments. In *Proceedings of the 4th ICSE Conference* (2001).
9. Kemper, A., and Neumann, T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of 27th ICDE* (2011), IEEE, pp. 195–206.
10. Leipzig, J. A review of bioinformatic pipeline frameworks. *Briefings in bioinformatics* (2016), bbw020.
11. Liu, T., Curtsinger, C., and Berger, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 327–336.
12. Majer, K. *Linux Kernel Internals, Second Edition.* Addison-Wesley, US, 1998.
13. McCracken, D. Sharing page tables in the linux kernel. In *Linux Symposium* (2003), p. 315.
14. Merrifield, T., and Eriksson, J. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 127–139.

15. Sikka, V., Färber, F., Goel, A. K., and Lehner, W. SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform. *PVLDB 6*, 11 (2013), 1184–1185.
16. Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 18–32.
17. Yu, J., and Buyya, R. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec. 34*, 3 (Sept. 2005), 44–49.
18. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., Franklin, M., Shenker, S., and Stoica, I. Fast and interactive analytics over hadoop data with spark. *USENIX Login 37*, 4 (2012), 45–51.