SI-CV: Snapshot Isolation With Co-Located Versions

Robert Gottstein, Ilia Petrov, Alejandro Buchmann

Databases and Distributed Systems Group, TU-Darmstadt, Germany

{gottstein | petrov | buchman}@dvs.tu-darmstadt.de

Abstract. Snapshot Isolation is an established concurrency control algorithm, where each transaction executes against its own version/snapshot of the database. Version management may produce unnecessary random writes. Compared to magnetic disks Flash storage offers fundamentally different IO characteristics, e.g. excellent random read, low random write performance and strong read/write asymmetry. Therefore the performance of snapshot isolation can be improved by minimizing the random writes. We propose a variant of snapshot isolation (called SI-CV) that collocates tuple versions created by a transaction in adjacent blocks and therefore minimizes random writes at the cost of random reads. Its performance, relative to the original algorithm, in overloaded systems under heavy transactional loads in TPC-C scenarios on Flash SSD storage increases significantly. At high loads that bring the original system into overload, the transactional throughput of SI-CV increases further, while maintaining response times that are multiple factors lower.

1 Introduction

Database systems, their architecture and algorithms are built around the IO properties of the storage. In contrast to Hard Disk Drives (HDD), Flash Solid State Disks (SSD) exhibit fundamentally different characteristics: high random and sequential throughput, low latency and power consumption [4]. SSD throughput is asymmetric in contrast to magnetic storage, i.e. reads are significantly faster than writes. Random writes exhibit low performance, which also degrades over time. Therefore, to achieve balanced performance, random writes should be avoided at the cost of random reads.

Snapshot Isolation (SI) is a Multi-Version Concurrency Control (MVCC) algorithm, in which every transaction operates against its own workspace/snapshot of the database. Under SI read operations do not block writes and vice versa, which is a good match for the Flash SSD properties. SI provides significant performance improvements compared to two-phase locking schedulers. Whenever a transaction modifies a tuple in its workspace a new version of that tuple is created and linked to the chain of older versions. Such operations result in undesired random writes. On algorithmic level no provisioning is made for this case. On system level SI relies solely on the buffer manager to intercept random writes.

We extended the classical SI algorithm to collocate/group tuple versions created by a transaction in the same or in adjacent database pages, employing a mechanism of page pre-allocation. We call the algorithm Snapshot Isolation with Co-located Versions (SI-CV).

The contributions of the paper are: (i) we implemented SI-CV in PostgreSQL (ii) SI-CV was tested in an OLTP environment with DBT2[6] (an open source version of TPC-C [8]) (iii) SI-CV performs up to 30% better than the original algorithm on flash SSDs and equally good on HDD; (iv) SI-CV performance is better under heavy loads; (v) SI-CV is space efficient, regardless of its pre-allocation mechanism.

The rest of the paper is organized as follows: in the following section we briefly review the related work; the properties of Flash SSDs are discussed in Section 3; then we introduce the original SI algorithm and SI-CV. Finally, section 6 describes the experimental results and analyses.

2 Related Work

The general SI-algorithm is introduced and discussed in [1]. The specifics of the PostgreSQL SI implementation are described in detail in [2,3]. As reported in [1] SI fails to enforce serializability. Recently a serializable version of SI was proposed [7] that is based on read/write dependency testing in serialization graphs. Serializable SI assumes that the storage provides enough random read throughput needed to determine the visible version of a tuple valid for a timestamp, making it ideal for Flash storage. [9] recently made an alternative proposal for SI serializability. In addition serializable SI has been implemented in the new (but still unstable) version of PostgreSQL and will appear as a standard feature in the upcoming release.

SI-CV presents a way to specially collocate data (versions) for each transaction leveraging the properties of the SSDs. There are several proposed approaches for flash storage managers, the majority of which explore the idea of append based storage [12] for SSDs [11, 10]. SI-CV differs in that it collocates per transaction but does not eliminate the concept of write in-place by converting all writes into

appends. Although it is our long term goal to integrate log-based storage mechanisms this is not part of this work.

In addition, there exist several proposals for page layouts [13] such as PAX [14] that aim at sorting row data in a column-based order with page sub-structures. Such approaches are developed within the context of Data Warehousing and show superior performance for read-mostly data. [15] explores how query processing algorithms and data structures such as FlashJoin [15] or FlashScan can benefit from such page organizations and the characteristics of Flash SSDs.

Furthermore, there have been numerous proposals of improving the logging and recovery mechanisms with respect to new types of memories (Flash SSDs, NVMemories). In-Page Logging [16] (IPL) is one such mechanism, that allows significant performance improvements by write reduction as well as page and log record collocation.

We expect that techniques such as Group Commit have a profound effect on version collocation approaches in MVCC environments. We have not explored those due to their effect on database crash recovery; however it is part of our future work. In [17] we explore the influence of database page size on the database performance on Flash storage.

In a series of papers, e.g. [2] Kemme et al. investigate database replication approaches coupled to SI. SI has been implemented in Oracle, PostgreSQL, Microsoft SQL Server 2005. In some systems as a separate isolation level, in others to handle serializable isolation.

To the best of our knowledge no version handling approaches for SI exist. This aspect has been left out of consideration by the most algorithms as well.

3 ENTERPRISE FLASH SSDS

The performance exhibited by Flash SSDs is significantly better than that of HDDs. Flash SSDs, are not merely a faster alternative to HDDs; just replacing them does not yield optimal performance. Below we discuss their characteristics.

(a) asymmetric read/write performance – the read performance is significantly better than the write performance – up to an order of magnitude (Fig. 1, Fig. 2). This is a result of the internal organization of the NAND memory, which comprises two types of structures: pages and blocks. A page (typically 4 KB) is a read and write unit. Pages are grouped into blocks of 32/128 pages (128/512KB). NAND memories support three operations: read, write, erase. Reads and writes are performed on a page-level, while erases are performed on a block level. Before performing a write, the whole block containing the page must be erased, which is a time-consuming operation. The respective raw latencies are: read-55µs; write 500µs; erase 900µs. In addition, writes should be evenly spread across the whole volume. Hence no write in-place as on HDDs, instead copy-and-write.

(b) excellent random read throughput (IOPS) – especially for small block sizes. Small random reads are up to hundred times faster than on an HDD (Fig. 1). The

good small block performance (4KB, 8KB) affects the present assumptions of generally larger database page sizes.

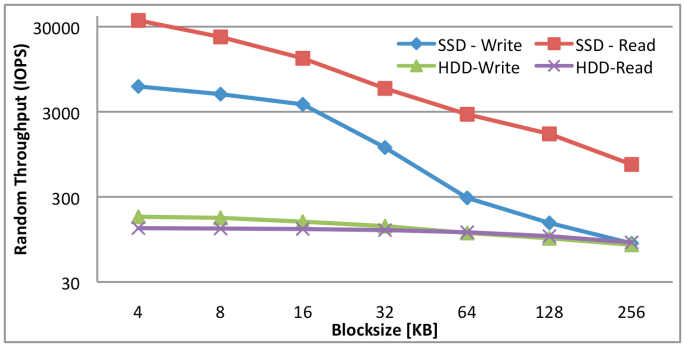


Figure 1. Random throughput (IOPS) of a X25-E SSD vs. HDD 7200 RPM

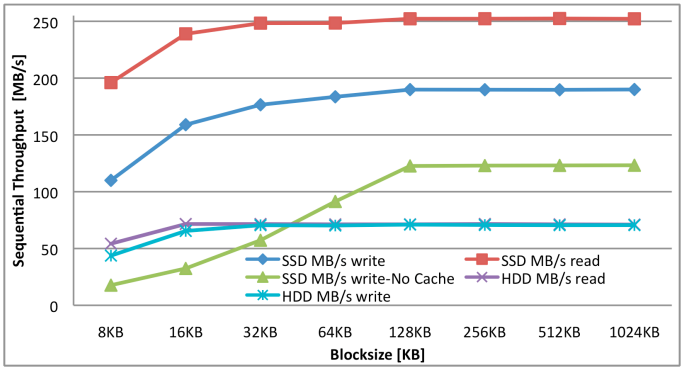


Figure 2. Sequential throughput (MB/s) of an X25-E SSD vs. HDD 7200 RPM

Blocksize 4 KB	Write Cache (WC)-ON				Write Cache-OFF	
	SSD		HDD		SSD	
	Avg[µs]	Max[ms]	Avg[ms]	Max[ms]	Avg[µs]	Max[ms]
Sequential Read	53	12.3	0.133	109.2	–	12.3
Sequential Write	59	94.8	0.168	36.9	455	100.3
Random Read	167	12.4	10.8	121	–	12.4
Random Write	113	100.7	5.6	127.5	435	100.7

Table 1. AVG/MAX latency of an X25-E SSD and 7200 RPM HDD

(c) low random write throughput – small random writes are five to ten times slower than reads (Fig. 1). Nonetheless, the random write throughput is an order of magnitude better than that of an HDD. Random writes are an issue not only in terms of performance but also yield long-term performance degradation due to Flash-internal fragmentation effects.

(d) good sequential read/write transfer (Fig. 2). Sequential operations are also asymmetric. However, due to read ahead, write back and good caching the asymmetry is below 25%.

4 SNAPSHOT ISOLATION

In SI [1] each transaction operates against its own version (snapshot) of the committed state of the database. If a transaction T_i reads a data item X, the read operation is performed from T_i's snapshot, which is unaffected by updates from concurrent transactions. Therefore, reads never block writes (and vice versa) and there is no need for read-locks. Modifications (inserts, updates, deletes - T_i's write set) are also performed on T_i's snapshot and upon successful commit become visible to appropriate transactions. During commit the transaction manager checks whether T_i's modifications overlap with the modifications of concurrent transactions. If write sets do not overlap T_i commits, otherwise it aborts. These commit-time checks are represented by two alternative rules: first-committer-wins[1] or first-updater-wins[1,3]. While the former is enforced in deferred manner at commit time, the latter results in immediate checks before each write. The first-updater-wins relies on write locks (see also Listing 1) and is implemented in PostgreSQL.

Apart from the general SI algorithm we also summarize its PostgreSQL implementation [1,2,3] (Listing 1). On begin of every new transaction it is assigned a unique transaction ID (TID) equivalent to a timestamp. Tuples in PostgreSQL are the unit of versioning. Every version V_i of a tuple X is annotated with two TIDs: *t_xmin* and *t_xmax*. *t_xmin* is TID of the transaction that created V_i. *t_xmax* is the TID of the transaction that created a new version V_j of X, a successor of V_i. In principle V_j invalidates V_i. If *t_xmax* is NULL,V_i is the most recent version. All versions are organised as a linked list in memory. Complementary to the tuple versions PostgreSQL maintains a *SnapshotData* structure for every running transaction T_i. Among other fields it contains: (i) *xmax* - the TID of the next transaction ($T_{(i+1)}$ at time T_i started) and serves as a visibility threshold for transactions whose changes are not visible to T_i; (ii) *xmin* – determines transactions whose updates are visible to T_i; (iii) *xip* – holds a TID list of all transactions concurrent to T_i. Finally PostgreSQL uses a main memory structure called PG_CLOG (previously pg_log), based on the database log, which allows for fast transaction status checks (aborted, committed, in progress).

Consider Listing1: whenever transaction T_i reads a tuple X (line 2), SI first checks if X is in the writeset of T_i to determine whether it has to read its own or the last stable version of X (line 2-4). Its own version can be read directly, because it cannot be modified by another transaction. Otherwise SI has to determine the version of X visible to T_i. Tuple visibility can be expressed with two conditions (line 19 and 20). The first one requires the X to be created by a transaction that successfully committed before T_i started. The second one (line 20) forbids X to be modified (and committed) by a concurrent transaction T_j.

Listing 1: Snapshot Isolation

```
1.  Start Transaction Ti → tsi = timestamp( Ti );
2.  ON Ti.read( X ):        // Transaction Ti reads tuple X
3.      IF( X IN { writeSet(Ti) } ) X.Vi = readOwnVersion( X,Ti )
4.      ELSE X.Vi  = readStableVersion( X, Ti )
5.  ON Ti.write( X ):       // Transaction Ti modifies tuple X
6.    IF( VersionCheck( X ) = FAILED) → Ti.rollback()
7.    Ti.lockX = requestXlock( X )
8.    IF( Ti.lockX == GRANTED ){       //PerformUpdate → InstallUpdate
9.      X.Vs = readStableVersion( X, Ti ); X.Vi =new Version(X);
10.     X.Vs.t_xmax = tsi;  X.Vi.t_min = tsi; X.Vi.t_max = NULL;
11.   } ELSE          //another transaction has already acquired the lock on X
12.     ENQUEUE( Ti.lockX ) → ... wait_for_lock ... ON lock granted
13.     GOTO line 6;//restart the write validation. avoid concurrent changes
14. ON Ti.commit() or Ti.rollback() →
15.   Release All acquired locks,Wake Up Waiting Transactions,Update Log
16. End Transaction Ti;

17. readStableVersion(Tuple X,Transaction Ti){
18.   Find X.Va, created by transaction Tj such that:
19.   // Find X.Va created by the latest Tj that committed before Ti started:
      X.Va.t_xmin |       X.Va.t_xmin < Ti.SnapshotData.xmax AND
                          PG_CLOG( X.Va.t_xmin ) == committed
20.   //Find X.Va that is untouched or was updated by Tj that either aborted
      // or was in progress when Ti attempted to write:
      X.Va.t_xmax |       X.Va.t_xmax==NULL OR
                          PG_CLOG( X.Va.t_xmax ) == aborted OR
                          X.Va.t_xmax IN {Ti.SnapshotData.xip}
21.   IF Checks FAIL return NULL ELSE return X.Va
22. }
23. VersionCheck( Tuple X ) {
24.   X.Vi = readStableVersion( X )
25.   IF( X.Vi == NULL) return FAILED // X was updated by concurrent Tj
26. }
```

Before T_i writes X, SI first performs a version check to determine if the version was updated by a concurrent transaction. On a negative check T_i has to abort (line 6). On a positive check it requests a write-lock on X. If X is locked by a concurrent transaction T_k, T_i waits until the lock is granted. Otherwise, it acquires a write lock on X, the stable version $X.V_s$ is read and a new version $X.V_i$ is created. Ti sets the creation timestamp $X.V_i.t\_xmin$ and the invalidation timestamp $X.V_s.t\_xmax$ to its own timestamp *tsi* (lines 9,10). On a commit or abort all acquired locks are released, waiting transactions are woken up and *PG_CLOG* is updated.

Snapshot Isolation never deletes an old version, however a tuple version may still become effectively invisible to any running transaction, because of the rules in line 19 and 20. Such obsolete versions consume precious space and can be safely removed. PostgreSQL runs a Vacuum process, which removes obsolete versions and coalesces free space. A simple version of Vacuum marks obsolete versions as deleted thus freeing space, while the exhaustive Vacuum version removes such versions and coalesces the freed space. Unfortunately, it requires an exclusive table lock and generates heavy I/O.

5 SNAPSHOT ISOLATION WITH CO-LOCATED VERSIONS

As Listing1 (lines 9,10) clearly shows, the present algorithm does not group the tuple versions created by transactions. It results into multiple updates (*t_xmax* of the old version and the newly created versions), which may lead to random writes. Our idea is to collocate all versions created by one transaction and group them into adjacent blocks. This not only minimizes the random writes, possibly converting them into sequential writes, but also uses potentially more random reads. Therefore SI-CV is a good match for SSD properties.

SI-CV (Fig. 1) introduces a new structure Barray in the database(shared) buffer of PostgreSQL. The Barray maps a transaction to a block-number. In SI-CV each transaction receives a pre-allocated block for inserts/updates of tuple versions, which is determined on the first write request. Read only or bulk-insert transactions have no entries in the Barray. With bulk inserts the storage manager writes sequentially, making it unnecessary to provision for that case. Upon the creation of the first new version of a tuple X by a transaction T_i an entry in Barray is created with an artificial block number. To determine a physical block SI-CV has to decide whether to use the free space map (FSM) or not. The FSM is a buffer manager structure that keeps track of block numbers that still have space left. If the FSM is used, an existing block with sufficient space for the new version is selected, which currently must not be in Barray. This is how we forbid multiple transactions to be mapped on the same block. However, one transaction may reserve multiple blocks in Barray. In absence of free space the FSM is not used, the relation is extended with a new block, which forms a new entry in the Barray structure together with the transaction ID. Upon transaction termination the entry in the Barray is deleted and made visible to the FSM. After the commit of a transaction, the buffer only contains committed data (versions).

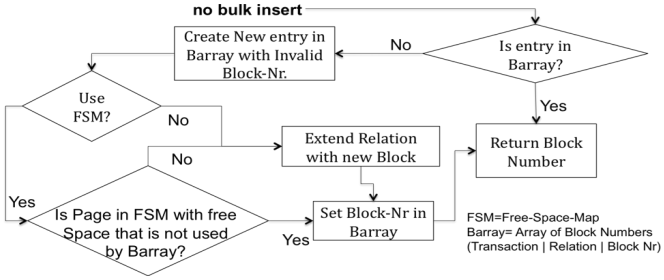


Fig. 1. SI-CV block diagram

We illustrate how SI-CV works based on a simple example (Fig. 2). Two transactions T_i and T_j modify the tuples K, R, X and Y in the relation Rel, in the following way: Start(T_i), Start(T_j), W_i[X], W_j[K], W_j[R], W_i[Y], Commit(T_i), W_j[K], W_j[R], Commit(T_j). According to SI-CV the Barray buffer manager structure will assign blocks uniquely to each transaction. These blocks will contain all versions created by the respective transaction (Fig. 2): transaction T_i with TID123 is mapped to Block1, while transaction T_j with TID124 is mapped to

Block2. Upon T_i's commit Block1 is written, upon T_j's commit Block2 is written. However, if T_j aborts Block2 is not written.

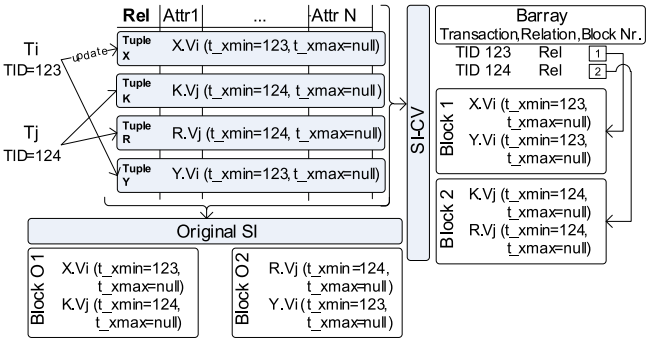


Fig. 2. SI-CV example

The original SI will pick any block with enough free space to host the new version, regardless of whether it hosts versions of other transactions. This undoubtedly yields random writes. In Fig. 2 block O1 and O2 contain versions from both transactions, which predisposes them to be written multiple times. Upon T_i's commit both blocks are written. Upon T_j's commit both blocks are re-written, because T_j overwrote the values of its own tuples K and R. However, if T_j aborts both blocks have to be re-written nonetheless.

Based on this example SI-CV not only minimizes on random writes and has better abort behavior, but should also perform better with higher number of transactions. We investigate this claim in the following section.

6 Evaluation

We implemented SI-CV using the PostgreSQL 8.4.2 codebase. The implementation spans several sub-modules of the storage manager, in particular the buffer and page mangers.

We tested SI-CV against the original SI algorithm on a machine with Intel Core 2 Duo 3GHz CPU and 512 MB RAM, running a 64-bit Ubuntu Server. In addition, we used an Intel X25-E/64GB enterprise SSD and a 7200RPM SATA2 HDD. The properties of both drives are described in Section 3. PostgreSQL is configured with a 24 MB shared buffer and activated simple vacuums (Section 4). The nominal DB size is 31GB. As benchmark we used DBT2[6], which is instrumented with 20 database connections and 20 terminals per warehouse. Every test run has a two hour duration, excluding the additional ramp-up time (which is proportional to the number of warehouses used).

Original SI [NoTPM]		SI-CV [NoTPM]			
SSD (270 Wh.)	HDD (80 Wh.)	SSD (270 Wh.)		HDD (80 Wh.)	
2500	210	3588	+30.3%	219	+3.8%

Table 2. Maximum DBT2 Transaction Throughput [NoTPM] with the respective number of warehouses

The DBT2 test results showing the maximum transaction throughput for SSDs and HDDs are displayed in Table 2. These show a performance increase of 30% with SI-CV on SSDs. SI-CV on HDDs performs slightly better with an improvement of 3.8%. The clear performance advantage of SI-CV on SSDs physically results from the reduction of random writes, at the cost of more random reads. As discussed in Section 3, both random operations have the same cost on a HDD, whereas random reads are much cheaper than random writes on a SSD. Hence the different rate of improvement (Table 2). In addition, a growing number of concurrent transactions, offers more room for version collocation, which magnifies the above effect.

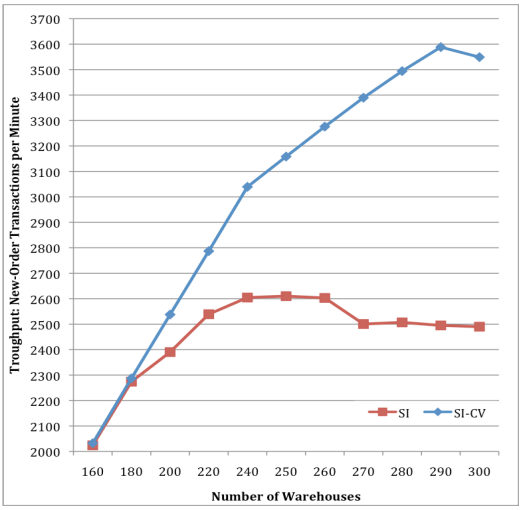


Figure 3. Transaction Throughput (New Order Transactions per Minute) SI-CV vs. SI

The performance effects of version collocation will increase with higher transactional loads. The reason for this is that more transactions create more versions of tuple data, which if collocated will save more random write operations.

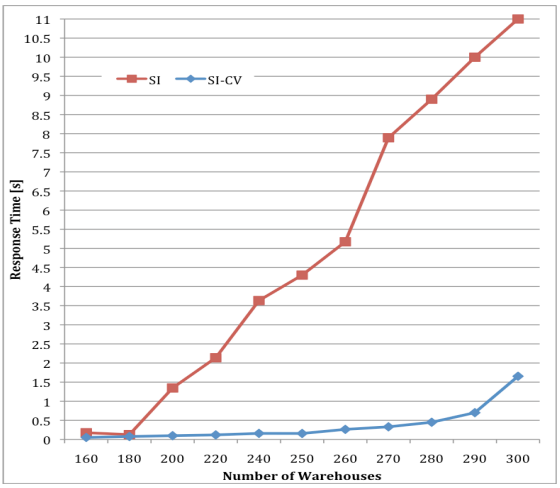


Figure 4. Avg. Resp. Time [s] of SI-CV and SI on SSD (lower is better)

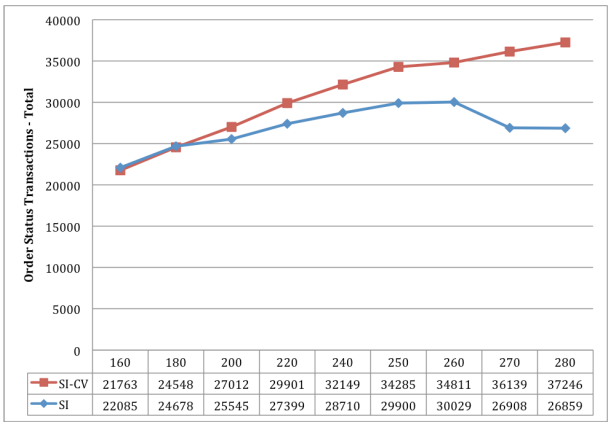
To verify this claim we performed a series of experiments, where the number of warehouses increases continuously thus producing higher transactional loads. (In TPC-C the number of transactions per warehouse is approximately constant – Section 5.2.3 from the TPC-C Specification [8] – hence increasing the number of warehouses increases the number of transactions).

The results in Figure 3 clearly show that SI-CV exhibits better performance under higher loads. On an under-committed system with enough free resources (Figure 3, Warehouses ≤ 180) SI and SI-CV perform equally well in PostgreSQL. A further increase of the load (Figure 3, Warehouses ≥ 230) brings SI into thrashing; the system is overloaded the transactional throughput does not increase further and begins to deteriorate, while the response times (Figure 4) increase exponentially. The throughput of SI-CV grows steadily for the same range of loads. SI-CV achieves up to 30% higher transactional throughput, before going into thrashing. Such performance behavior is especially favorable to whenever peak loads need to be processed or load spikes occur in real systems.

Another interesting characteristic of SI-CV are the low response times. As Figure 4 shows, on an under-committed system both SI and SI-CV have similar response times. SI-CV, however, can support higher transactional loads at significantly lower response times. For peak loads (in the present testbed; Warehouses ≥ 230), SI-CV provides up to 30% higher transactional throughput at sub-second response times (Figure 3 and Figure 4).

Furthermore, SI-CV offers similar or better read performance than the original SI. To verify this statement we report the ORDER_STATUS transaction performance, which is a read-only transaction. Figure 5 shows the total number of executed ORDER_STATUS transactions with SI and SI-CV for each two hour test run with different number of warehouses. In this experiment all other TPC-C transactions (read-write) execute concurrently. The goal is to obtain a realistic mixture of reads and writes that according to the canonical SI do not affect each

other. The numbers in Figure 5 show that read performance of SI-CV remains unaffected by the version collocation changes.



	160	180	200	220	240	250	260	270	280
SI-CV	21763	24548	27012	29901	32149	34285	34811	36139	37246
SI	22085	24678	25545	27399	28710	29900	30029	26908	26859

Figure 5. Total number of Order Status Transactions

Last but not least, we report the disk space consumption of SI-CV, for the following reason. The price per GB of disk space on an enterprise 15K RPM HDD is ~7x lower than on an enterprise SSD. Due to the block pre-allocation per new transaction these blocks may not be filled optimally: each SI-CV block may contain more unused space than an SI block. After a two hour TPC-C test the space consumed by SI-CV increased by less than 0.0001% per Warehouse compared to the original SI. Hence, SI-CV is almost as space efficient as the SI.

Conclusions

We developed an extension of Snapshot Isolation (SI), called Snapshot Isolation with Co-located Versions (SI-CV). It places versions of tuples created or modified by a transaction in pre-allocated blocks. Thus it reduces the amount of random writes, which leverages better the properties of Flash SSDs. SI-CV is implemented in PostgreSQL.

TPC-C tests show that:

(a) SI-CV performs better especially under heavy load conditions where the system is very I/O-bound. Under such conditions we achieved up to 30% better performance with SI-CV.

(b) The relative performance of SI-CV (to SI) increases with higher number of transactions.

(c) The transaction response time with SI-CV on an over-committed system remains significantly lower than that of SI. Under heavy load conditions SI-CV operates with sub-second response times.

(d) SI-CV utilizes a block pre-allocation strategy per transaction. We prove experimentally that it is almost as space efficient as SI. The space consumption difference is **marginal** and justifies the performance advantages of SI-CV.

(e) Finally, the read performance of SI-CV in comparison to SI is equally good or better.

Acknowledgments

The authors wish to thank Todor Ivanov for his kind assistance with setting up the experimental environment. This work was supported by the DFG project "Flashy-DB".

References

[1] [1] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P. 1995. A critique of ANSI SQL isolation levels. In Proc. The ACM SIGMOD'95 (San Jose, California, United States, May 22 - 25, 1995).

[2] Wu, S., B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. Proc. of the IEEE ICDE, Tokyo, Japan, 2005.

[3] Korth, H., A. Silberschatz. Database System Concepts. McGraw-Hill Publishing Company, 2001.

[4] Chen, F., Koufaty, D. A., Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In Proc. of SIGMETRICS '09 (Seattle, WA, USA), 2009

[5] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., Panigrahy, R. Design tradeoffs for SSD performance. In USENIX08 Boston, Massachusetts, June 2008.

[6] Database Test Suite. DBT2. http://osdldbt.sourceforge.net/

[7] Cahill, M. J., Röhm, U., Fekete, A. D. 2008. Serializable isolation for snapshot databases. In Proc. SIGMOD 2008 (Vancouver, CA, 2008)

[8] TPC Benchmark C. Standard Specification. Revision 5.11. Feb. 2010 http://www.tpc.org/tpcc/spec/tpcc_current.pdf

[9] Revilak, S. ; O'Neil, P. ; O'Neil, E.. Precisely Serializable Snapshot Isolation (PSSI). Data Engineering (ICDE), 2011 IEEE 27th International Conference on. 11-16 April 2011

[10] P. Bernstein, C. Rein, and S. Das. Hyder -- A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[11] Stoica, R., M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN '09) . 2009

[12] M. Rosenblum, J. Ousterhout. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. 10, 1 (February 1992), 26-52.

[13] A. Ailamaki, D. DeWitt, M. Hill. Data page layouts for relational databases on deep memory hierarchies. The VLDB Journal 11, 3 (November 2002), 198-21. 2002

[14] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. VLDB, pages 169–180, 2001.

[15] D. Tsirogiannis, S. Harizopoulos, M. Shah, J. Wiener, G. Graefe. Query processing techniques for solid state drives. In Proceedings of the 35th SIGMOD international conference on Management of data (SIGMOD '09) 2009

[16] S.-W. Lee, B. Moon. Design of flash-based DBMS: an in-page logging approach. In Proceedings of the 2007 ACM International conference on Management of data (SIGMOD '07) . 2007

[17] I. Petrov, R. Gottstein, T. Ivanov, D. Bausch, A. Buchmann. Page Size Selection for OLTP Databases on SSD RAID Storage. In Journal of Information and Data Management,Vol.2,No 1 2011