MV-IDX: Multi-Version Index in Action

Robert Gottstein #1, Rohit Goyal #1, Ilia Petrov #2, Sergey Hardock #1, Alejandro Buchmann #1

#1Databases and Distributed Systems Group, TU-Darmstadt, Germany,
{gottstein, goyal, hardock, buchmann}@dvs.tu-darmstadt.de

#2 Data Management Lab, Reutlingen University
ilia.petrov@reutlingen-university.de

Abstract: Multi-Versioning DBMS (MV-DBMS) represent a very good match to the properties of Flash storage and the combination of both offers conceptual advantages. Yet, the specifics of indexing in MV-DBMS on Flash have been widely neglected. Although an index in a MV-DBMS references multiple versions of a data item, it is only allowed to return a single (at most one) version of that data item "visible" to the current index operation. Logically separating version visibility checks from the index structure and operations, as in the traditional version-oblivious index, leads to version management overhead: to determine the appropriate version of a data item, the MV-DBMS first fetches all versions that match the search criteria and subsequently discards invisible versions according to the visibility criteria. This involves unnecessary I/Os to fetch tuple versions that do not need to be checked. We propose the idea that version-aware indexing has additional responsibility to recognize different tuple versions of a single data item and to filter invisible tuple versions in order to avoid unnecessary I/Os. In this work we demonstrate an approach called Multi-Version Index (MV-IDX) that allows index-only visibility checks which significantly reduce the amount of I/O as well as the index maintenance overhead. MV-IDX is implemented in the PostgreSQL open source MV-DBMS. We demonstrate that the MV-IDX achieves significantly lower response times and higher transactional throughput on OLTP workloads than the version-oblivious approach. We showcase latency and throughput improvements by utilizing the DBT2 TPC-C benchmarking tool and report saved I/Os. We also showcase how the proposed approach performs better on SSDs.

1 Motivation

MV-DBMS require management of tuple versions of a data item. This has conceptual advantages for Flash [GPB13]: (i) with multi-versioning write operations never block reads, which matches the high read-performance and intrinsic parallelism of Flash; (ii) updates of data items create new tuple versions that are separate physical entities which can be algorithmically utilized to eliminate random writes and ultimately provide write-sequentialisation. However multi-versioning brings up the specific issue of visibility: out of the set of all tuple versions at most one version can be visible to a transaction. Database indices need to handle additional visibility aspects ([JJ07]). [HJS+09] describes a general index structure for MV-DBMS. Traditional (version oblivious) indexing forces the MV-DBMS to determine the visibility after each matching version was fetched (Fig. 1), since

such approaches store visibility information physically on each version. In addition the index needs to contain records referencing all existing tuple versions. Visibility can therefore only be determined with additional I/O access [GGH+14]. Since version management is handled by the MV-DBMS concurrency control, the index cannot optimize accesses to invisible tuple versions.

In this demonstration we present the Multi-Version Index (MV-IDX), a version-aware index structure that is capable of answering visibility decisions efficiently in-memory, solely by accessing the index and logically working on two dimensions: the indexed attribute and the data item. Independently of the underlying physical storage layout, a tuple's visibility can be determined before it is fetched from disc, thus reducing the overall I/O to the storage device. Furhtermore index management overhead is avoided on updates that do not change the search-key value. The MV-IDX is implemented into the PostgreSQL MV-DBMS, using *Snapshot Isolation* concurrency control with the *first-updater-wins* rule. We showcase latency and throughput improvements using OLTP workload generated by the TPC-C benchmark and compare it to the unaltered conventional indexing within PostgreSQL. We show that the MV-IDX achieves significantly lower response times as well as higher throughput on OLTP workload (Fig. 2). In addition we present the amount of saved I/O accesses by the MV-IDX. The audience can vary the workload using the TPC-C/DBT2 parameters. In addition we offer specific microbenchmarks to stress special workloads on the MV-IDX.

2 The Multi-Version Index

Figure 1 depicts the basic structures of the demonstrated MV-IDX algorithm. The MV-IDX uses a virtual identifier (VID) that uniquely identifies a data item and all indexed tuple versions belonging to it. Instead of a tuple identifier (TID) the VID is stored in the index: the typical MV-IDX index record comprises a *Key-VID* pair, where the traditional index records comprise *Key-TID* pairs. A single data item is therefore identified by a VID and each tuple version by its TID.

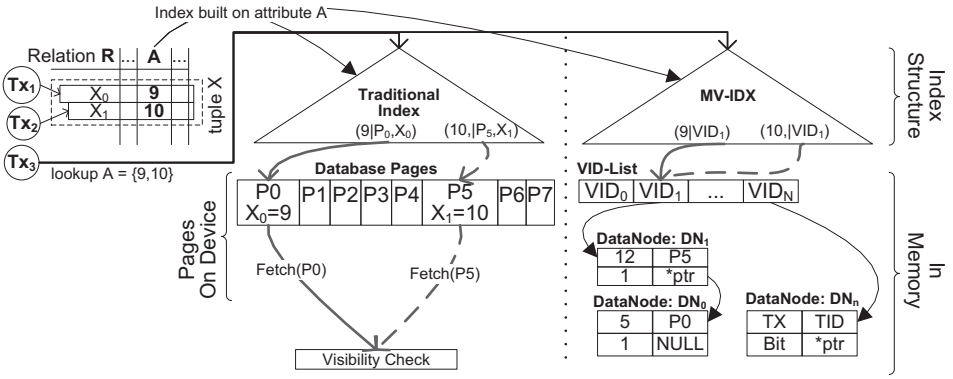

Figure 1: MV-IDX Example and Major Data Structures

Data Structures The MV-IDX uses two in-memory data structures (Fig. 1):

(i) The *VID-List* mapping structure that stores a pointer for each VID to a single *data node*. The pointer always points to the data node that describes the newest (most recent) tuple version. (ii) One *Data Node* per tuple version, comprising: *a) the transaction timestamp* (TX) and *b) the TID* of the tuple version; *c)* a *Bit* that indicates whether the timestamp (a) denotes the commit time of the tuple version or its insertion time (e.g., the inserting transaction is still active) and *d) a pointer* $*ptr$: If a potentially visible tuple version directly preceding the most recent one exists, the pointer refers to the *data node* that stores information about it; otherwise a *NULL* value is stored.

Size and Management The size of the in-memory data structures is deterministic and reasonable, the $VID - List$ has to store n data node pointer, where n denotes the amount of date items in the relation. The size of each node sums up to 136bit: the transaction ID (TX, 32bit), tuple ID (TID, 64bit), status bit (Bit) and the pointer ($*ptr$, 32bit). Each entry in the $VID - List$ uses 32bit. Data nodes that describe tuple versions which are not visible to any running transaction any more can be garbage collected.

Algorithm: Fig. 1 depicts an example of the MV-IDX to clarify its principle, details are provided in [GGH+14]. Relation R has an index on attribute A. Consider transaction TX_1 that *inserts* a new data item X and thus creates the first tuple version X_0. X receives the unique VID_1, identical among all tuple versions X_i of X. A Data Node DN_0 is created, containing the VID (VID_1), TID of the new tuple version ($P0$) and the transaction id of the creating transaction TX_1 (TX) are set. The bit is set to 0 to indicate that the tuple version is not yet committed; **ptr* is set to *NULL* since a preceding version does not exist. A pointer directing to DN_0 is stored in *VID-List*. The *Key-VID* pair is inserted at the appropriate position within the MV-IDX.

If a concurrently running transaction TX_{read-1} (not depicted) performs an index lookup that matches the key value of the new data item, the VID is retrieved and using the $VID - List$, the data node DN_0 is accessed. The status Bit in DN_0 is reset, indicating that the tuple version is not yet committed (TX shows that the transaction is concurrently running) - the visibility check therefore returns *Not-Visible*. TX_1 commits and finishes at time 5, the bit in DN_0 is set to 1 and the TX is set to 5. TX_{read-2} (not depicted) starts and requests to read X. The access on the MV-IDX leads to DN_0. The status bit indicates that the version is committed and visible since TX_{read-2} is greater than TXin DN_0. Updates proceed analogously to inserts. A new data node DN_1 is created (depicted in Fig. 1 as TX_2) that describes the new tuple version (VID, TID and TX fields are set accordingly). The pointer in *VID-List* is set to refer to DN_1 and the **ptr* on DN_1 refers to DN_0. Older transactions follow the $*ptr$ reference to DN_0.

So far TX_1 inserted data item X in version X_0, committed at transactional time 5, TX_2 creates the update X_1 and commits at transactional time 12. The value of attribute A changed and the index contains both versions of X. Note: If the indexed key value has not been changed nothing has to be updated or stored additionally in the MV-IDX - the *VID-List* points to the new data node.

TX_3 executes a lookup of all data items with values $A = \{9, 10\}$. The traditional index fetches each version, contained in two different pages from disc, resulting in 2 disc I/Os.

Both versions are checked for visibility - X_0 is discarded. The MV-IDX retrieves the VID and finds the only visible tuple version X_1 without the need for any disc I/O - independent of the underlying storage device (SSD, HDD). The visible tuple version can be fetched afterwards. The approach is also applicable to multicolumn searchkeys.

3 Demo Scenario

In our first demo scenario we showcase the MV-IDX implementation in PostgreSQL (Version 9.3.4). We execute testruns of the TPC-C benchmark using the open source DBT2 implementation. The results are compared to the unaltered PostgreSQL DBMS. The audience can alter the duration of the benchmark as well as the amount of warehouses. Varying these parameters shows the influence of the amount of tuple versions per data item, results are depicted in Fig. 2. The MV-IDX improves on the native implementation especially on longer testruns where the amount of tuple versions increases.

In our second demo scenario we execute microbenchmarks that directly point out the influence of a high amount of tuple versions per data item. Long running transactions are mixed with short running transactions in order to stress the management functions of the in-memory data structures to demonstrate their efficiency in access speed and memory consumption. In addition we sum up the amount of saved I/O accesses by the MV-IDX.

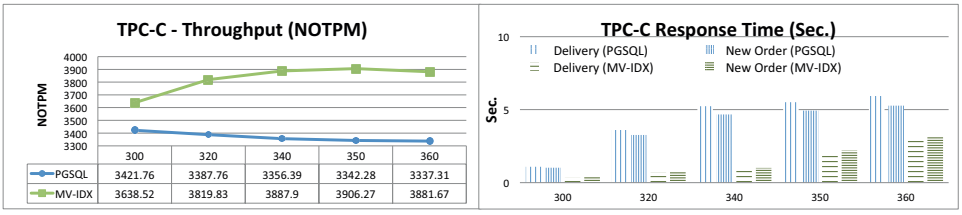

Figure 2: TPC-C on SSD: Throughput - Latency - 2h Runtime - 15 Clients - 150 to 360 Warehouses

Acknowledgments This work was supported by the DFG (Deutsche Forschungsgemeinschaft) project "Flashy-DB".

References

[GGH+14] Robert Gottstein, Rohit Goyal, Sergej Hardock, Ilia Petrov, and Alejandro Buchmann. MV-IDX: indexing in multi-version databases. In *IDEAS'14*. ACM, 2014.

[GPB13] Robert Gottstein, Ilia Petrov, and Alejandro Buchmann. Append storage in multi-version databases on flash. In *Big Data*. Springer Berlin Heidelberg, 2013.

[HJS+09] Tuukka Haapasalo, Ibrahim Jaluta, Bernhard Seeger, Seppo Sippu, and Eljas Soisalon-Soininen. Transactions on the multiversion B+-tree. In *Proc.*, EDBT '09, 2009.

[JJ07] Khaled Jouini and Geneviève Jomier. Indexing multiversion DBs. In *Proc CIKM*, 2007.