Efficient Logging for Enterprise Workloads on Column-Oriented In-Memory Databases

Johannes Wust
Hasso Plattner Institute
johannes.wust@hpi.uni-potsdam.de

Joos-Hendrick Boese
SAP AG
j.boese@sap.com

Frank Renkes
SAP AG
frank.renkes@sap.com

Sebastian Blessing
Hasso Plattner Institute
sebastian.blessing@hpi.uni-potsdam.de

Jens Krueger
Hasso Plattner Institute
jens.krueger@hpi.uni-potsdam.de

Hasso Plattner
Hasso Plattner Institute
hasso.plattner@hpi.uni-potsdam.de

ABSTRACT

The introduction of a 64 bit address space in commodity operating systems and the constant drop in hardware prices made large capacities of main memory in the order of terabytes technically feasible and economically viable. Especially column-oriented in-memory databases are a promising platform to improve data management for enterprise applications. As in-memory databases hold the primary persistence in volatile memory, some form of recovery mechanism is required to prevent potential data loss in case of failures. Two desirable characteristics of any recovery mechanism are (1) that it has a minimal impact on the running system, and (2) that the system recovers quickly and without any data loss after a failure. This paper introduces an efficient logging mechanism for dictionary-compressed column structures that addresses these two characteristics by (1) reducing the overall log size by writing dictionary-compressed values and (2) allowing for parallel writing and reading of log files. We demonstrate the efficiency of our logging approach by comparing the resulting log-file size with traditional logical logging on a workload produced by a productive enterprise system.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems

Keywords
Databases; Column Store; In-Memory; Logging

1. INTRODUCTION

The separation of transactional (OLTP) and analytical data management (OLAP) as found in today’s enterprises complicates the implementation of modern business applications, which cannot be classified as either transactional or analytical, but integrate queries from both worlds. In-memory database management systems (IMDBMS) that store data in dictionary-compressed columns, have been proposed to consolidate OLTP and OLAP workloads in a single system [12]. To use IMDBMS in productive enterprise applications, they need to provide the same durability guarantees as existing OLTP and OLAP systems, as well as at least comparable performance.

Like in a disk-based DBMS, durability in an IMDBMS is achievable by writing log information to backup storage, such as persistent memory. However, touching backup storage for every transactional write operation potentially creates a severe bottleneck, since write performance of the IMDBMS is limited by the throughput of I/O channels. Besides the problem of impeding transactional write operations, logging strategies must also allow for fast recovery of the primary in-memory copy to increase availability of the DBMS.

The main strategy to increase transactional write performance as well as recovery speed is to apply multiple I/O channels and disks. However, writing logs in parallel induces the problem that the specific order of data changes is not reflected in the log data anymore, since log entries are scattered around multiple disks/volumes. This becomes a significant problem for recovering dictionary-compressed data, as the order of inserted values determines the dictionary code. Recreating the order, e.g., by writing timestamps or loading and reordering the log data in-memory at recovery time is prohibitively expensive.

To address the aforementioned problems for column-oriented IMDBMS, we propose a transaction-safe logging scheme for dictionary-compressed columns that (1) reduces the size and number of log information written to disk, (2) allows writing log data to multiple log volumes in parallel, and (3) enables reading and processing log information in parallel at recovery time. The key idea to achieve these goals is to store data dictionary-compressed and decouple dictionary inserts from the transactional context.

The speed of logging and recovery in a parallel environment depends heavily on the hardware architecture, number of I/O channels and bandwidth, but is largely independent of the content of written data. Hence, we demonstrate the gen-
eral possibility for parallel logging and recovery (Sections 4 and 5), and focus the experimental validation in Section 6 on the produced log size for different workloads, which is directly affected by our logging scheme.

2. RELATED WORK

Related work in the area of logging and recovery can be found for traditional, disk-based databases, as well as recovery specific to in-memory databases.

With a focus on disk-based systems, Haerder and Reuter [4] present a terminology for describing different transaction-oriented recovery schemes. Commercial state-of-the-art DBMS, such as Oracle’s 11g and IBM’s DB2, allow for parallel log writes and parallel recovery to decrease start-up time. A detailed description of Oracle’s physiological log file format, as well as parallel log writing and recovery can be found in [15].

Research that focuses on checkpointing and crash recovery for memory resident databases is provided in [3, 13], and [10], as well as more recent in [1]. We consider work on checkpointing complimentary to our work, as we rely on regular checkpointing in our system model.

Specific on logging for in-memory databases, Lee et. al [9] propose a commutative and associative logging scheme for highly parallel in-memory database systems. In contrast to our work we propose a logging scheme specifically designed for dictionary-compressed columns.

Looking at other recently developed main memory databases, we find that traditional logical logging is applied in HyPer [6]. On the other hand, H-store [14] relies on multiple replicas and recovery from a remote site. A comparison of data replication for recovery and log-based recovery can be found in [8].

3. SYSTEM MODEL

In this Section, we present a logical system architecture of the IMDBMS underlying this work, emphasizing on the data structures that need to be recovered in case of a failure. Figure 1 shows the relevant components of the IMDBMS, focusing on the components involved in logging and recovery. A detailed description of the assumed system architecture can be found in [12].

For each column of a table, we maintain a vector of value IDs and a corresponding dictionary with a mapping of values to value IDs, whereas the value ID is implicitly given by the row ID of the dictionary column.

To increase read performance and utilize main memory more efficiently, a part of this primary data is stored in a read-optimized format [7], while changes are accumulated in a write-optimized structure. A snapshot of the read-optimized structure is written to persistent storage, whenever data from the write-optimized structure is merged into the read-optimized store. For the logging mechanism described here, we assume the read-optimized part to be restored from snapshots identified by checkpoints in our log data and focus on the logging and recovery of the write-optimized data.

For every transactional write operation, the Log Manager writes log messages to I/O-Buffers, which are eventually flushed to disk when a transaction commits. In case a buffer gets flushed, all entries in this buffer are written in a log to persistent memory. We keep separate logs for the column entries, $L_{vu}$, as well as the dictionaries, $L_{di}$, as described in more detail in Section 4. We assume that the IMDBMS can utilize multiple I/O channels to write log data in parallel, while a dedicated write buffer, i.e. for individual disks or SSDs, can be used for each log partition. In case of a failure, the Recovery Manager uses these logs to rebuild the in-memory data structures. The Consistent View and Transaction Manager are involved in transaction handling as discussed in [12]. We apply a multi version concurrency control (MVCC) scheme to determine which records are visible when multiple transactions run in parallel.

The process of handling write operations and all involved data structures is further illustrated in Figure 2 and explained in much greater detail in [12]. In case of a system failure, we need to recover the database to a consistent state. Following [4], a database is in a consistent state “if and only if” it contains the results of all committed transactions”. Changes to the database by partially executed and not committed transactions are not recovered. To recover the database to a consistent state after a failure, we need to recover (1) the order and the value IDs of the Value ID vector of each column, (2) the order and the values of the corresponding dictionary, and (3) the order and values of the corresponding system attributes.

4. DC LOGGING

This section describes our logging mechanism for dictionary-compressed column structures, referred to as DC Logging (for Dictionary-compressed Columns). The main design goals for our logging mechanism are to allow for parallel log writing and recovery, as well as a reduction of the overall log volume. The key idea to achieve these goals is to store values dictionary-compressed. As the dictionary mapping is updated each time a new value is inserted, purely logical log records would have to be replayed in the original sequential order to reconstruct the value IDs, preventing parallel logging and recovery. Hence, we decouple logs for dictionary mappings and value IDs and log dictionary updates outside of the transactional context. This way, the values ID vectors and dictionaries can be written and recovered independently and in parallel.

4.1 Structure of Logs

In this section, we introduce the different logs required to recover the data structures described in Section 3. These are
dictionary, value, and transaction logs. We further describe meta data which is logged to speed up recovery.

Dictionary logs $Lv$ are written each time a transaction writes a new value into a column. To allow the recovery of this information, we need to log the table name $t$, the column index $c_i$ of the column the dictionary belongs to, the value $v$ added to the dictionary, as well as the corresponding value ID $VID$. Hence, a dictionary log is defined as follows, while "d" identifies the log a as dictionary log:

$$Ld = \{d, t, c_i, v, V ID\}$$

Notice that we do not log the transactional context of a dictionary entry. As $Ld$ contains the $VID$ and $v$, dictionary logs can be replayed in any order while $v$ can be inserted at position $VID$ in the dictionary vector during recovery. Value logs $Lv$ are used to restore the value ID vectors of a table, as well as the system attributes indicating the invalidated row of a transaction. They are written for every record added to the write-optimized structure, identified by a flag "$v$", and contain the following attributes:

$$LD = \{v, TID, t, RID, IRID, bm_v, (VID_1, ..., VID_n)\}$$

The row ID $RID$ and value IDs $VID_i$ identify the changed attributes. Additionally, we need to log the affected table $t$ and the transaction ID $TID$ of the transaction manipulating the table, as well as the row ID of a potentially invalidated row for updates or deletes, the IRID.

To reduce the size of the log entry, we only log the $VID$s for attributes that are actually changed by the update, but not for all attributes of the record. Therefore, a bit mask $bm_v$ is stored, indicating which attributes have been changed by the operation. During recovery, the missing attribute values of a row can be derived from its logical predecessor record as described in Section 5. To identify the value logs by the TID that need to be recovered, we have to write a log entry once a transaction has been committed. Transaction logs $Lt$ have the following structure:

$$Lt = \{t', TID\}$$

Transaction logs are written after all corresponding dictionary and value logs have been flushed to disk. Figure 2 shows an example for the resulting log entries of three transactions. The resulting changes to table T1 are indicated by the corresponding shadings. Notice that for transaction TA2 the updated value 'M' does not lead to a dictionary log, as it is already in the dictionary. Also, notice that the value log of transaction TA2 does not contain any value ID of the attribute Attr1, which is not changed in the UPDATE statement.

To speed up recovery, we propose to additionally log metadata. This prevents to move data in memory during recovery, since the required memory can be reserved in one block at recovery start. Metadata that allows speeding up recovery is: (i) table size, i.e., number of records in a table; and (ii) the number of bits used in dictionary encoding.

### 4.2 Synchronization of Log Writes

Transaction logs $Lt$ indicate that a transaction with $TID$ is committed. To guarantee consistent recovery, the transaction log of a committed transaction is written to disk after all corresponding value logs have already been persisted. Furthermore, to guarantee that no value log is recovered that references to a value ID, which is not covered by a corresponding dictionary log, the dictionary log buffer must be flushed to disk before the commit log of the transaction that inserted the dictionary mapping is written.

As introduced in Figure 1, we have dedicated buffers for value logs, as well as dictionary logs. Transaction logs are written in the same log buffer as value logs. If we have multiple IO-buffers, value log buffers can be partitioned by $TID$, and dictionary log buffers by table name or column index. Once a transaction is about to commit, all dictionary log buffers dedicated to the dictionaries of columns in which values have been inserted during this transaction have to be flushed first. Then a transaction log is written to the respective value log buffer for its $TID$ and this buffer is flushed.

## 5. RECOVERY

In this section, we sketch a proof that the data structures introduced in Section 3 can be consistently recovered from the logs, and further describe the parallel recovery process.

### 5.1 Verification of Consistent Recovery

As described in Section 4.2, a transaction $TA_i$ is committed by writing a transaction log $Lt$ if all corresponding dictionary and value logs of $TA_i$ have been written to persistent storage. Hence, the changes of a committed transaction $TA_i$ can be consistently recovered if all corresponding value and dictionary logs can be found in persistent storage. All value logs $Lv_1, ..., Lv_l$ of a transaction $TA_i$ are flushed to disk once the transaction commits, as they are written in the same log buffer as the transaction log. Dictionary logs are not handled within a transactional context; hence, we have to prove that if a transaction $TA_i$ commits, all dictionary entries relevant for $TA_i$ have been flushed to disk, too.

We can distinguish three cases for dictionary logs: (1) a value written by transaction $TA_i$ is not present in the dictionary. (2) a value written by transaction $TA_i$ is already present in the dictionary and has been flushed to persistent memory, and (3) a value written by transaction $TA_i$ is already present in the dictionary, but has not been flushed to persistent memory.

In case (1), a dictionary log for the new value is written to a log buffer which is flushed once the transaction commits (see Section 4.2). In case (2) the relevant dictionary entry is already stored in persistent memory. In case (3) another transaction $TA_j$ that is not yet committed has written a
dictionary log to the corresponding log buffer; this buffer is flushed as soon as the next transaction, that changed a value in the corresponding column commits (see Section 4.2), at the latest when TA commits and independent whether TA eventually commits or aborts. Therefore, all dictionary entries relevant for TA, have been flushed to persistent storage once TA commits.

5.2 Parallel Recovery

In the first step of a recovery run, the meta log file is read to determine the latest available snapshot of the read-optimized store and approximate size of the primary data to be recovered in main memory. As soon as the TID of the last successfully snapshotted transaction for a table is found, the according snapshot is directly restored from SSD/disk.

After reading the meta log, replaying of value log files to recover the write-optimized store can start in parallel to recovering the read-optimized structures. A value log file is processed in reverse order, i.e., the latest log entry is read first. We do this, as the last log entry of a committed transaction is always the transaction log with state commit. By processing the log file in reverse order, we can determine the TIDs of all committed transactions to find out which value and consistent view logs we need to consider for recovery. A value log is applied only if a transaction log with a corresponding TID was encountered previously.

Dictionary logs can be applied in parallel to value log replay, as they are independent of the transactional context. As described in Section 4.1, we log only changed value IDs for update operations, which leads to incomplete rows in a table. Therefore, we need to complete the records in a second run over the data structures in memory.

Algorithm 1 describes the process of completing records for a table from the logical predecessor record. Logical predecessors are linked via the invalidated row field of each record. We iterate over the table starting with the first record — the first record is always complete. When we approach the first record that has empty fields, all predecessors are complete. Hence, we can complete all missing records by filling them with the according records of the predecessor (p in Algorithm 1). We iterate once over the entire table and once over the fields of each empty row.

Algorithm 1 Completing Records

for i := 0 → records.length do
    r ← records[i]
    if r has empty fields then
        i ← invalidated row ID of r
        p ← records[i]
        for all fields m in r do
            n ← accordant field of m in p
            if m == nil then
                m ← n
            end if
        end for
    end if
end for

6. SIMULATION OF LOG SIZES

In this Section, we discuss drivers of the log size for our DC logging scheme and demonstrate its applicability for enterprise workloads by simulating resulting log sizes for different workloads. Therefore, we implemented a simulator that maintains the data structures as described in Section 3, including table structures with columns and respective dictionaries. Our simulator parses a given SQL workload and generates the log entries described in Section 4. Besides our proposed DC logging mechanism, we implemented record- and consistent view logs we need to consider for recovery. A value log is applied only if a transaction log with a corresponding TID was encountered previously.

For value sizes, the difference in overall log size between logical and DC logging becomes more significant for larger values, as logical logging stores the same value several times. When varying the number of distinct values, a larger number leads to higher log sizes in DC logging, as more dictionary logs need to be stored. In the extreme case of 100% distinct values, the log size of our logging would exceed logical logging due to the overhead of the dictionary logs.

6.1 Influencing Parameters of Log Size

To demonstrate the impact of various characteristics and conclude for which workloads and data characteristics our logging scheme performs best, we have simulated the resulting log size for a million inserts of single zipf distributed values with a total of 1000 distinct values and compared this to logical logging. We have applied the mean alpha value of 1.518 for the zipf distribution as identified in the analyzed enterprise data in [5].

The results of this analysis show that our logging scheme performs best if (1) large values, such as long strings, are inserted multiple times, and (2) the number of distinct values is low. These characteristics are typical for enterprise data as shown in [5].

For value sizes, the difference in overall log size between logical and DC logging becomes more significant for larger values, as logical logging stores the same value several times. When varying the number of distinct values, a larger number leads to higher log sizes in DC logging, as more dictionary logs need to be stored. In the extreme case of 100% distinct values, the log size of our logging would exceed logical logging due to the overhead of the dictionary logs.

6.2 Logging Database Workloads

This section demonstrates the performance of our logging mechanism using a widely accepted benchmark workload, TPC-C [11], as well as a workload from a productive SAP ERP installation.

To simulate a TPC-C workload, we ran a DBT-2 benchmark [2] for one warehouse for 60min on a MySQL database and logged the executed SQL statements. Then we fed our simulator with the approximately 24,000 queries on tables with a total of roughly 600,000 records. The DBT-2 workload contains a mix of INSERT and UPDATE statements on various tables with attributes of types integer, float, time stamp and string. We see a high number of distinct values, mainly as most data is generated randomly in DBT-2, which is unfavorable for the proposed DC logging as discussed in Section 6.1. For most columns, the resulting value distribution differs significantly from the enterprise characteristics we have seen in the analyses of enterprise data [5].

Figure 3 shows the result of the TPC-C workload for logical logging, as well as for our proposed DC logging. In total, our logging scheme produces around 10% less log volume. We see that the dictionary log size contributes around 58%
of the total DC log size.

To validate the applicability of DC logging for enterprise workloads, we have reconstructed a workload from the change logs of a productive SAP ERP installation. Figure 4 shows the graph for the cumulated log size per query of 7 million write operations on the sales item table for logical logging, as well as for DC logging. Overall, DC logging reduces the log size for this particular case by 29%. It is noticeable, that the average log size per query decreases rapidly during the first 100,000 queries, with a local peak at roughly 150,000. When analyzing the write operations, we see that the first insert statements, as well as the insert statements after 150,000 queries, include inserts of large new values. For DC logging, this mainly affects the dictionary logs, as value logs only store a value ID independent of the value size. Furthermore, we see the average log size per query for logical logging increasing at around 1.5 million queries, but no similar effect for the DC logs. Analyzing the workload, we see that the average size of the inserted values has increased. As these values have already been stored in the dictionary, this causes no increase in average log size per query for DC logging. When comparing the results of the enterprise workload in Figure 4 with the results from the TPC-C Benchmark in Figure 3, we see that the dictionary log is a much smaller fraction of the total DC log size. Within TPC-C, mainly randomly generated string values drive the log size, which we do not see in the enterprise workload, where longer string values such as product names or addresses are often repeated.

7. REFERENCES


```plaintext
Figure 3: Cumulated average log size per query for tpc-c benchmark queries
```

```plaintext
Figure 4: Cumulated average log size per query for a real enterprise system workload
```