

Prof. Hasso Plattner



A Course in In-Memory Data Management

The Inner Mechanics
of In-Memory Databases

September 4, 2015

This learning material is part of the reading material for Prof. Plattner's online lecture "In-Memory Data Management" taking place at www.openHPI.de. If you have any questions or remarks regarding the online lecture or the reading material, please give us a note at openhpi-imdb@hpi.uni-potsdam.de. We are glad to further improve the material.

Research Group "Enterprise Platform and Integration Concepts",
<http://epic.hpi.uni-potsdam.de>

Logging

To be used in productive enterprise applications, databases need to provide durability guarantees (as part of the ACID¹ principle). To provide these guarantees, fault-tolerance and high availability have to be ensured. However, since hardware failures or power outages cannot be avoided or foreseen, measures have to be taken which allow the system to recover from failures.

The standard procedure to enable durable recovery is logging. With the help of logging and recovery protocols, databases can be brought back to the last consistent state before the failure. This is achieved by checkpointing the current system and logging subsequent data modifications. The data is written into log files, which are stored on persistent memory such as hard disk drives (HDD) or solid-state drives (SSD).

Please note that these requirements are true for any database, regardless of being an in-memory database or not.

29.1 Logging Infrastructure

A key consideration when talking about logging is performance, both for writing the logs as well as for reading logs back into memory when recovering. As discussed in Section 4.6, the performance gap between disk and CPU is steadily increasing. Consequently, logging has to be primarily optimized with respect to minimizing I/O operations.

Figure 29.1 outlines the logging infrastructure of SanssouciDB. The logging data, which is written to disk, consist of three parts:

- Snapshot of the main store
- Value logs

¹ ACID stands for Atomicity, Consistency, Isolation, Durability. These properties guarantee reliability for database transactions and are considered as the foundation for reliable enterprise computing.

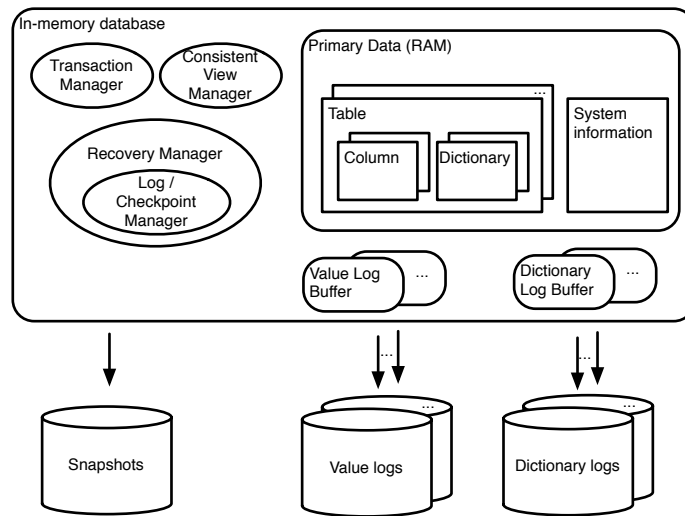


Fig. 29.1: Logging Infrastructure

- Dictionary logs

Check pointing [Bor84] is used to create a snapshot of the database at a certain point in time, at which the data is in a consistent state. According to [HR83], a database is in a consistent state “if and only if it contains the results of all committed transactions”. The snapshot is a direct copy of the read-optimized main store and it is written to disk periodically. The purpose of check pointing is to speed up the recovery process, since only log entries after the snapshot have to be replayed, while the main store can be loaded from the snapshot directly. To log the data of the differential store, which is not part of the snapshot, value logs as well as dictionary logs are used to track committed changes.

SanssouciDB’s logging infrastructure differs from most traditional databases. SanssouciDB adapted the infrastructure to leverage the columnar data structures and to reduce I/O performance penalties. Amongst these optimizations are:

- **Snapshot format:** at each checkpoint, a snapshot of the main store is directly written to disk in binary format. This means that an exact copy of the main store in memory is written to disk, which can later be directly restored without any transformation overhead in case of a recovery.
- **Checkpoint timing:** the ideal timing for check pointing is when the differential buffer is relatively small compared to the main store. That is right after the merge.
- **Storing meta data:** to speed up the recovery process, additional meta data is written to disk. With the help of this meta data, the required memory

can be allocated before loading. Thus, expensive re-allocations and data movements can be avoided. Data hereby written is, e.g., the number of tuples in the main store and the number of bits used in each dictionary.

- **Separation into value and dictionary logs:** the two major performance optimizations for logging in SanssouciDB are the reduction of the log size and the parallelization of logging. This is achieved by using dictionary-encoded logging, which is discussed in detail in the next section.

29.2 Logical vs. Dictionary-Encoded Logging

The obvious way to log data modifications is logical logging. As depicted in Figure 29.2, logical logging simply writes the SQL statement and its parameters (recordID and attribute values) to disk.

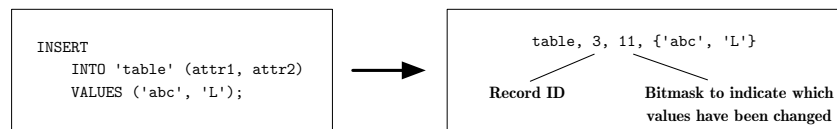


Fig. 29.2: Logical Logging

Logical logging has two major shortcomings. First, logging and recovery cannot be parallelized since the order of the log has to be preserved during replay to recover the dictionary and the corresponding attribute vector elements. Second, logical logging writes values directly to disk and, therefore, does not leverage compression as used in SanssouciDB. Consequently, logical logging writes comparatively large data volumes to disk.

To avoid these shortcomings, SanssouciDB uses the so-called dictionary-encoded logging schema [WBR⁺12], which separates the dictionary-encoded data and their corresponding dictionary inserts from the transactional context. This approach allows writing and recovering of the attribute vectors and dictionaries in parallel. Furthermore, dictionary-encoded logging reduces the log size due to the usage of dictionary-compression, which speeds up the recovery process significantly.

In which cases dictionary-encoded logging is advantageous over logical logging depends on the data characteristics. In enterprise applications, the same data characteristics that favor dictionary-compressed column-stores also apply to dictionary-encoded logging. Amongst these characteristics are, e.g., a low number of distinct values, which leads to fewer dictionary log entries, and the distribution of values.

Many real world value distributions can be described using the Zipf distribution. Intuitively, the Zipf distribution describes – depending on the

variable *alpha* – how heavily the distribution is drawn to one value. In the case of $\alpha = 0$ the distribution equals a uniform distribution and every value occurs equally often. As α increases, fewer values occur more frequently (see exemplary distributions for varying α values in Figure 29.3).

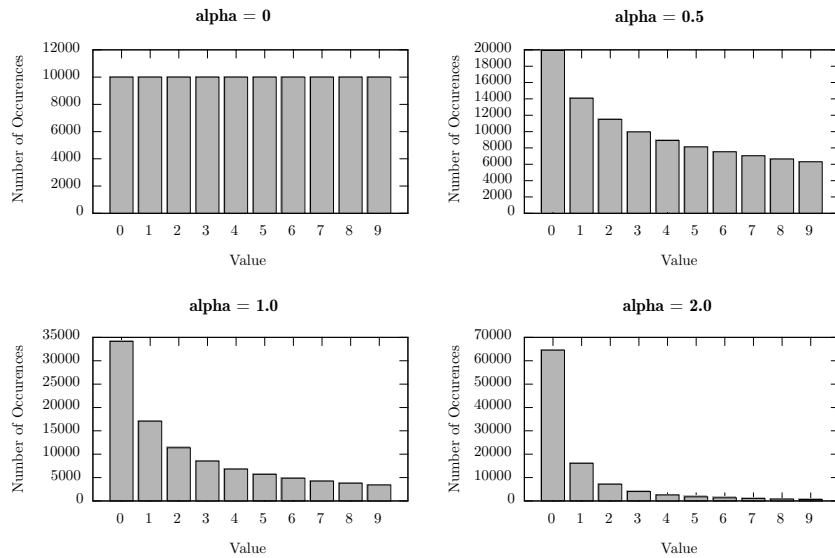


Fig. 29.3: Exemplary Zipf Distributions for Varying Alpha Values

Hübner et al. [HBK⁺11] state that the majority of columns analyzed from financial, sales and distribution modules of an enterprise resource planning (ERP) system were following a power-law distribution – a small set of values that occur frequently, while the majority of values is rare. Furthermore, they identified an average α value of 1.581 in enterprise systems.

Figure 29.4 shows the results of an experiment that measures the cumulated log size per query for a varying value distribution. In this experiment, one million INSERT queries of 1,000 distinct zipf distributed values have been simulated. With an α value of 1.581, the dictionary is already saturated after $\approx 30,000$ inserts. Afterwards, queries rarely add entries to the dictionary log. As shown for an α value of 4.884 (see Figure 29.4), the more heavily the distribution is drawn to one value, the smaller the accumulated log size will be. Please note that logging is only needed for queries that modify the data set to save the changes.

A comparison of the log sizes for logical and dictionary-encoded logging is shown in Figure 29.5. These values have been measured on a productive enterprise system with seven million write operations on the sales item table. Due to the high compression of recurring values, the dictionary-encoded log-

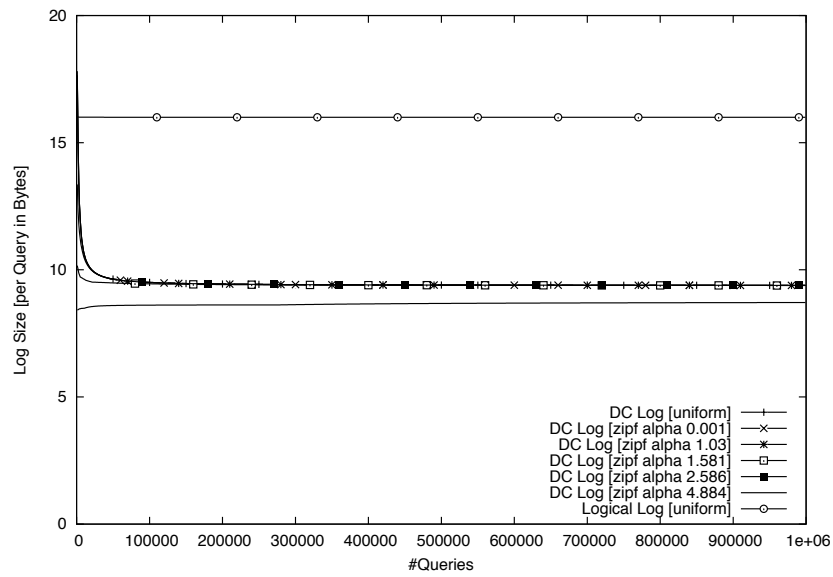


Fig. 29.4: Cumulated Average Log Size per Query for Varying Value Distributions (DC = Dictionary-Compressed)

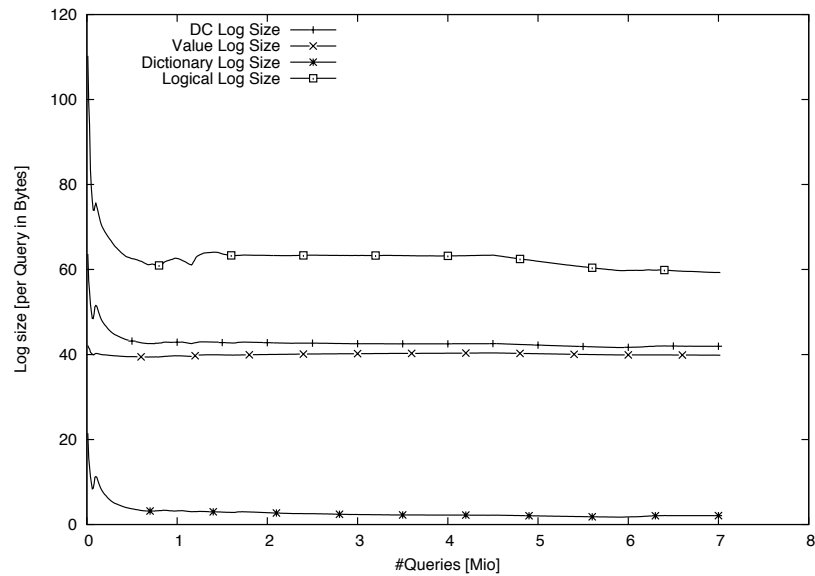


Fig. 29.5: Log Size Comparison of Logical Logging and Dictionary-Encoded Logging

ging reduces the log size by 29 %. As a consequence, dictionary-encoded logging is in favor of logical logging, since it exploits typical data distributions in enterprise systems. For more details about enterprise data characteristics see Chapter 3.

29.3 Example

Figure 29.6 shows an example of dictionary-encoded logging. Here, three SQL queries (insert, update, and delete) of three different transactions are logged.

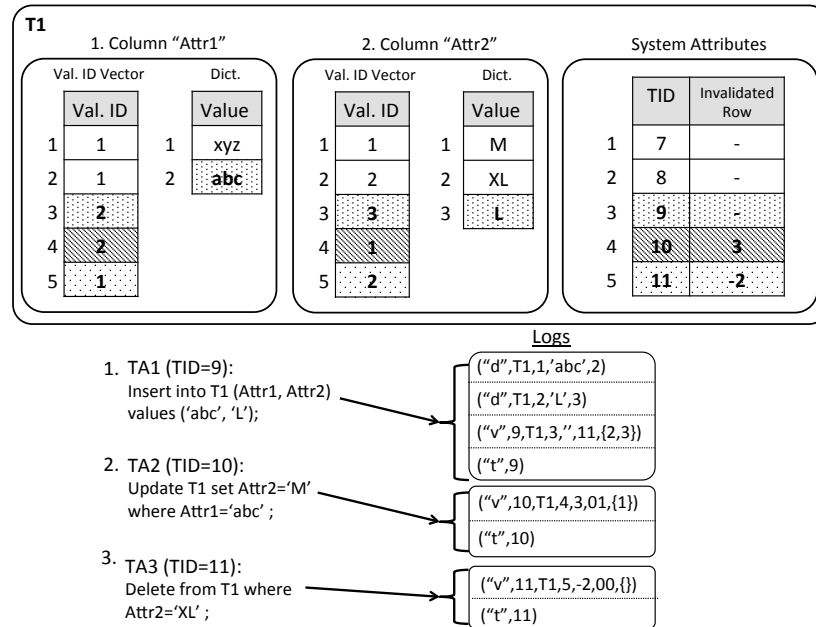


Fig. 29.6: Example: Logging for Dictionary-Encoded Columns

The first statement (INSERT INTO T1 (Attr1, Attr2) VALUES ('abc', 'L')) inserts a new row into table T1. This statement has the transaction ID 9 (TID=9), which is stored in the following format:

$$L_t = \{ "t", TID \}$$

Since both values ('abc' and 'L') are not yet stored in the corresponding dictionaries, new entries will be added. Dictionary logs L_d are created each

time a transaction adds new values into a dictionary. Therefore, the table identifier t , the column index c_i , the added value v , and the corresponding valueID VID are logged. The letter “d” at the first position of the dictionary log entry marks that this is a dictionary log entry, similar to the letter “t” marking the transaction log entry above.

$$L_d = \{“d”, t, c_i, v, VID\}$$

Consequently, for transaction 9, two dictionary logs are stored holding the IDs of the modified dictionaries for table T1 (i.e. column ID), the corresponding positions in the dictionaries, and the newly inserted values:

$$(“d”, T1, 1, ‘abc’, 2) \text{ and } (“d”, T1, 2, ‘L’, 3)$$

Value logs L_v store the actual values, which are appended to the attribute vectors. Value log entries store more than just plain data structure changes as done in the dictionary logs, since they have to be linked to the corresponding transactions.

$$L_v = \{“v”, TID, t, RID, IRID, bm_n, (VID_1, \dots, VID_n)\}$$

Each value log L_v hereby stores the transaction identifier TID , the table identifier t , and the rowID RID in the attribute vector. The letter “v” at the first position marks this log entry as a value log entry. The values are stored in a vector of VID s, whereby the bit mask bm_n marks the corresponding columns (n is the number of attributes in table t). If a row is invalidated by the new row (e.g. due to an update or a delete), the ID of the invalidated row is stored in $IRID$.

The second statement (`UPDATE T1 SET Attr2 = ‘M’ WHERE Attr1 = ‘abc’`) in Figure 29.6 alters a row, without introducing new values. Thus only one transactional log entry and one value log entry, storing the new dictionary position for attribute “Attr1”, are written to disk.

The delete statement (`DELETE FROM T1 WHERE Attr2 = ‘XL’`) invalidates a row. But in this case, the result might not be obvious. When a row is deleted, a new line is added to the table. This is necessary to reflect the changes made by the transaction in the system attributes of the table. In this example, transaction 11 marks row 2 as invalid. This is achieved via hidden system attributes (i.e. columns storing the TID and the corresponding IDs of invalidated rows). While the TID field of a certain row always persists the transaction ID that inserted this row, the invalidated row field is only written for updates and deletes. To mark that a line has not just been updated but entirely deleted, the invalidated row field is prefixed (see entry “-2” in Figure 29.6). Both, for updates and deletes, the unchanged fields of the inserted tuple are copied from the invalidated row instead of being left empty. The reason is twofold: first, with fixed length attribute vectors, empty fields provide no advantages in terms of performance or memory consumption.

Second, copying the row avoids additional lookups to get the values of the invalidated row. This is especially advantageous for long-running transactions and queries, which potentially need to include outdated rows.

It is furthermore important to understand when and in which order the stored logs are written to disk. Once a transaction shall be committed, first the dictionary buffers have to be written to disk. This has to be ensured to avoid value logs referencing valueIDs that cannot be recovered. Afterwards, the value logs are written to disk. Finally, if both logs were written to disk successfully, the committed transaction log is written to disk. Both, the value log entries and the transaction log entries are collected in the same log buffer. This minimizes the amount of flushes to be done upon transaction completion.

To replay the log, the logs are parsed in reverse order (i.e. from the newest to the oldest entry). This is done to ensure that only transactions are written into the differential buffer that have been committed successfully as their commit logs are read before the dictionary and value logs when reading in reverse order.

29.4 References

- [Bor84] Andrea J. Borra. Robustness to crash in a distributed database: A non shared-memory multi-processor approach. In Umeshwar Dayal, Gunter Schlageter, and Lim Huat Seng, editors, *VLDB*, pages 445–453, San Francisco, CA, USA, 1984. Morgan Kaufmann.
- [HBK⁺11] Florian Hübner, Joos-Hendrik Böse, Jens Krüger, Cafer Tosun, Alexander Zeier, and Hasso Plattner. A Cost-Aware Strategy for Merging Differential Stores in Column-Oriented In-Memory DBMS. In *BIRTE*, pages 38–52, 2011.
- [HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.
- [WBR⁺12] Johannes Wust, Joos-Hendrik Boese, Frank Renkes, Sebastian Blessing, Jens Krueger, and Hasso Plattner. Efficient Logging for Enterprise Workloads on Column-Oriented In-Memory Databases. In *CIKM 2012*. ACM, 2012.