SanssouciDB: An In-Memory Database for Processing Enterprise Workloads

Hasso Plattner
Hasso-Plattner-Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
Email: hasso.plattner@hpi.uni-potsdam.de

Abstract: In this paper, we present SanssouciDB: a database system designed for serving ERP transactions and analytics out of the same data store. It consists of a column-store engine for high-speed analytics and transactions on sparse tables, as well as an engine for so-called combined columns, i.e., column groups which are used for materializing result sets, intermediates, and for pocessing transactions on tables touching many attributes at the same time. Our analysis of SAP customer data showed that the vast majority of transactions in an ERP system are of analytical nature. We describe the key concepts of SanssouciDB's architecture: concurrency control, techniques for compression and parallelization, and logging. To illustrate the potential of combining OLTP and OLAP processing in the same database, we give several examples of new applications which have been built on top of an early version of SanssouciDB and discuss the speedup achieved when running these applications at SAP customer sites.

1 Introduction

The motto for the last 25 years of commercial DBMS development could well have been "One Size Fits All" [SMA+07]. Traditional DBMS architectures have been designed to support a wide variety of applications with general-purpose data management functionality. All of these applications have different characteristics and place different demands on the data management software. The general-purpose database management systems that rule the market today do everything well but do not excel in any area.

Directly incorporating the characteristics of certain application areas and addressing them in the system architecture as well as in the data layout can improve performance by at least a factor of ten. Such major gains were reported from database systems tailored to application areas such as text search and text mining, stream processing, and data warehousing [SMA+07]. In the following, we will use the term characteristic-oriented database system to refer to such systems. Our vision is to unite operational processing and analytical processing in one database management system for enterprise applications. We believe that this effort is an essential prerequisite for addressing the shortcomings of existing solutions to enterprise data management and for meeting the requirements of tomorrow's enterprise applications (see also [Pla09]).

This paper introduces SanssouciDB, an in-memory database for processing enterprise workloads consisting of both transactional and analytical queries. SanssouciDB picks up the idea of a characteristics-oriented database system: it is specifically tailored to enterprise applications. Although the main constituents of SanssouciDB's architecture are well-known techniques which were previously available, we combine them in a novel way.

The paper is organized as follows: Section 2 gives an overview of SanssouciDB's architecture. Afterwards, we describe three important components of the architecture in greater detail: Section 3 provides the reader with details of how data access is organized in main memory and how compression weighs in. Section 4 describes transaction management in SanssouciDB by explaining how concurrency control is realized as well as presenting the techniques used for logging and recovery. In Section 5 we present the parallel aggregation and join algorithms that we implemented for SanssouciDB. We believe that SanssouciDB's architecture has a great potential for improving the performance of enterprise applications. Therefore, in Section 6, we give a couple of application examples where significant improvements could be achieved at a number of SAP customer sites using the concepts presented in this paper. Section 7 concludes the paper.

2 Architecture of SanssouciDB

Nearly all enterprise applications rely on the relational data model, so we have made SanssouciDB a relational database system. The relations stored in SanssouciDB permanently reside in main memory, since accessing main memory is orders of magnitude faster than accessing disk. Figure 1 presents a conceptual overview of SanssouciDB. SanssouciDB runs on a cluster of blades in a distributed fashion, with one server process per blade. The server process itself can run multiple threads, one per physical core available on the blade, managed by a scheduler (not shown in Figure 1).

To communicate with clients and other server processes, a SanssouciDB server process has an interface service and a session manager. The session manager keeps track of client connections and the associated parameters such as the connection timeout. The interface service provides the SQL interface and support for stored procedures. The interface service runs on top of the distribution layer, which is responsible for coordinating distributed metadata handling, distributed transaction processing, and distributed query processing. To allow fast, blade-local metadata lookups, the distribution layer replicates and synchronizes metadata across the server processes running on the different blades. The metadata contains information about the storage location of tables and their partitions. Because data may be partitioned across blades, SanssouciDB provides distributed transactions and distributed query processing. The distribution layer also includes the transaction manager. While there are many interesting challenges in the distribution layer, we omit a detailed discussion of these topics in this paper. Data replication for column-oriented databases is discussed in [SEJ$^+$11].

The main copy of a database table is kept in main memory (rather than on disk) and

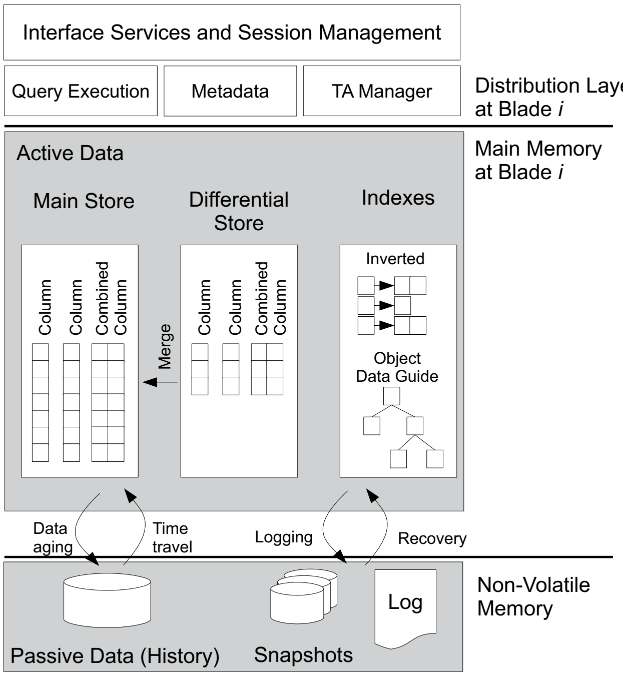


Figure 1: Conceptual Overview of SanssouciDB

consists of a main store, a differential store, and a collection of indexes. Non-volatile storage is required to provide the persistence for the database. Section 3.1 presents a detailed discussion about the separation into main and differential store.

Ideally, we would like to fit the complete database of an enterprise onto a single blade, that is, into a machine with a single main board containing multiple CPUs and a large array of main memory modules. However, not even the largest blades available at the time of writing allow us to do so. We thus assume a cluster of multiple blades, where the blades are interconnected by a network (see Figure 1).

A necessary prerequisite for a database system running on such a cluster of blades is data partitioning and distribution across blades. Managing data across blades introduces more complexity into the system, for example, distributed query processing algorithms accessing partitions in parallel across blades have to be implemented, as we will describe in Section 5. Furthermore, accessing data via the network incurs higher communication costs than blade-local data access. Finally, different data partitioning strategies have an impact on query performance and load balancing. Therefore, from time to time, it can become necessary to reorganize the partitions to achieve better load balancing or to adapt to a particular query workload. Some of our considerations on data placement and dynamic cluster reorganization can be found in [SEJ+11].

After deciding on a multi-blade system as the target hardware, the next question is: should many less powerful low-end blades be used or do we design for a small number of more powerful high-end blades? For SanssouciDB, we chose the latter option, since high-end blades are more reliable and allow more blade-local data processing thereby avoiding expensive network communication to access data on remote blades. In our target hardware configuration, a typical blade contains up to 2 TB of main memory and up to 64 cores. With 25 of these blades, we can manage the enterprise data of the largest companies in the world.

To make efficient use of this architecture, SanssouciDB exploits parallelism at all levels. This includes distributed query processing (among blades), parallel query processing algorithms (among cores on a blade) and exploiting Single Instruction Multiple Data (SIMD) instructions at processor level [WPB+09].

Combined columns as shown in Figure 1 are column groups in the sense of fine-grained hybrid data layout [GKP+11], which will be discussed in Section 3.1. Column grouping is particularly advantageous for columns that often occur together as join or group-by columns (see also the aggregation and join algorithms presented in Section 5). In the following sections, we will examine the concepts shown in Figure 1 in greater detail.

3 Data Access and Compression

In this section, we describe how SanssouciDB organizes data access in main memory and how compression is used to speed up processing and make efficient use of the available main memory capacity.

3.1 Organizing and Accessing Data in Main Memory

Traditionally, the data values in a database are stored in a row-oriented fashion, with complete tuples stored in adjacent blocks on disk or in main memory. This allows for fast access of single tuples, but is not well suited for accessing a set of values from a single column. The left part of Figure 2 exemplifies this by illustrating the access patterns of two SQL statements: the result of the upper statement is a single tuple, which leads to a sequential read operation of an entire row in the row store. However, accessing a set of attributes leads to a number of costly random access operations as shown in the lower left part. The grey shaded part of the memory region illustrates data that is read, but not required. This happens as data is read from main memory in chunks of the size of a cash line which can be larger than the size of a single attribute.

An analysis of database accesses in enterprise applications has shown that set-based reads are the most common operation [Pla09], making row-oriented databases a poor choice for these types of applications. Column-oriented databases [ABH09], in contrast, are well suited for these set-based operations. In particular, column scans, where all the data values that must be scanned are read sequentially, can be implemented very effi-

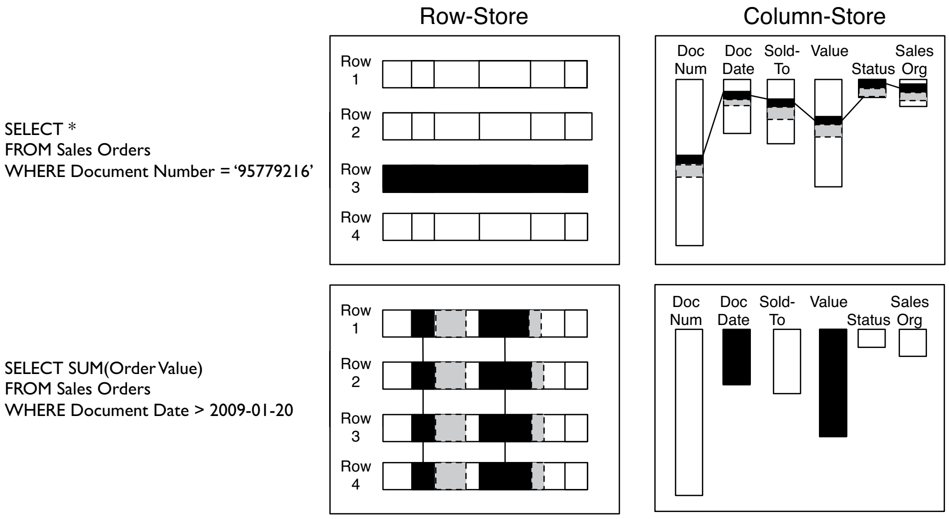
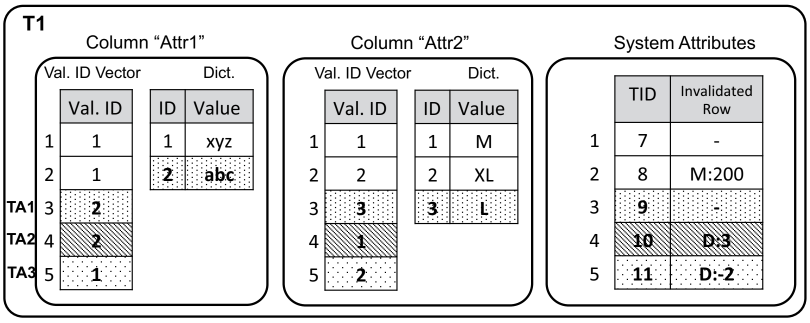


Figure 2: Operations on the Row Store and the Column Store

ciently. The right part of Figure 2 illustrate these considerations. The different lengths of the columns illustrates a varying compression rate; compression is described later in this section. Good scan performance makes column stores a good choice for analytical processing; indeed, many commercial column-oriented databases target the analytics market, for example, SAP Business Warehouse Accelerator and Sybase IQ. The disadvantage of column-oriented databases is that the performance of row-based operations is typically poor. To combine the best of both worlds, SanssouciDB allows certain columns to be stored together, such as columns that are frequently queried as a group. In the following, we refer to these groups of columns as combined columns (see Figure 1). Allowing these column types combines the advantage of the column-oriented data organization to allow for fast reads with good write performance. Further benefits of these combined columns are described in Section 5.

As outlined above, column stores provide good scan performance. To evaluate a query predicate on a column, for example, finding all occurrences of a certain material number, a column scan is applied. However, when the query predicate has a high selectivity, that is, when only a small number of all rows are returned, scanning results in too much overhead. For columns that are often queried with highly selective predicates, like primary or foreign key columns, SanssouciDB allows the specification of inverted indexes (see Figure 1).

To reduce the need for locking and to allow us to maintain a history of all changes to the database, we adopt an insert-only approach. We define the term "insert-only" as follows: An insert-only database system does not allow applications to perform updates or deletions on physically stored tuples of data. In SanssouciDB, all write operations insert a new tuple into the differential buffer, while the main store is only accessed by read operations. To track the different versions of a tuple, a table in the differential buffer contains two



Write Transactions:

```
TA1 (TID=9):     INSERT into T1 (Attr1, Attr2) values ('abc', 'L');
TA2 (TID=10):    UPDATE T1 set Attr2='M' where Attr1='abc';
TA3 (TID=11):    DELETE FROM T1 where Attr2='XL';
```

Figure 3: Column Store Write Operations

system attributes for each record: the *TID* of the transaction that wrote the record and an *invalidated row* field referencing to the row that became invalid by inserting this record, i.e., the previous version of the record. In case of an insert operation this field is left empty. Figure 3 depicts an example of insert and update operations and their effect on the differential buffer, for example, TA_2 updates row 3 and inserts D:3 into the invalidated row field to signal that row 3 of the differential buffer is now invalid and is the successor of the record in row 4.

A consequence of this insert-only approach is that data volumes increase over time. Our objective is to always keep all the relevant data in main memory, but as new data is added over time this becomes increasingly difficult. To ensure low latency access to the most recent data we make use of data aging algorithms to partition data into active data, which is always kept in main memory, and passive data that may be moved to flash-based storage, if necessary. The history store, which is kept in non-volatile storage, is responsible for keeping track of passive data. Keeping the history allows SanssouciDB to execute time-travel queries, which reflect the state of the database at any user-specified point in time.

3.2 Compression

As main memory sizes have grown rapidly, access latency to main memory has become the bottleneck for the execution time of computations: processors are wasting cycles while waiting for data to arrive. This is especially true for databases as described in [ADHW99]. While cache-conscious algorithms are one way to improve performance significantly [ADH02, RR00, RR99], another option is to reduce the amount of data transferred from and to main memory, which can be achieved by compressing [WKHM00]. On the one hand, compression reduces I/O operations between main memory and CPU registers, on the other hand, it leverages the cache hierarchy more effectively since more information fits into a cache line.

The number of CPU cycles required for compressing and decompressing data and the savings in CPU cycles from shorter memory access time result in increased processor utilization. This increases overall performance as long as the database system is I/O bound. Once compression and decompression become so CPU-intensive that the database application is CPU bound, compression has a negative effect on the overall execution time. Therefore, most column-oriented in-memory databases use light-weight compression techniques that have low CPU overhead [AMF06]. Common light-weight compression techniques are dictionary encoding, run-length encoding, bit-vector encoding, and null suppression.

In SanssouciDB, we compress data using dictionary encoding. In dictionary encoding, all values of a column are replaced by an integer called value ID. The original values are kept in a sorted array called dictionary. The value ID is the position of the value in the dictionary; see Figure 4. Our experiments have shown than run-length encoding on a sorted column incurs the fewest amount of cache misses among these compression techniques. However, applying run-length encoding requires sorting of each column before storing it. In order to reconstruct records correctly, we would have to store the original row ID as well. When reconstructing records, each column must be searched for that ID resulting in linear complexity. As enterprise applications typically operate on tables with up to millions records, we cannot use explicit row IDs for each attribute in SanssouciDB, but keep the order of attributes for each tuple identical throughout all columns for tuple reconstruction. Dictionary encoding allows for this direct offsetting into each column and offers excellent compression rates in an enterprise environment where many values, for example, country names, are repeated. Therefore, dictionary encoding suits our needs best and is our compression technique of choice in SanssouciDB. Read performance is also improved, because many operations can be performed directly on the compressed data. For a more complete discussion of compression, we refer the reader to [LSF09].

3.3 Optimzing write operations

As described in the previous section, data is compressed to utilize memory efficiently. This causes write operations to be expensive, because they would require reorganizing the storage structure and recalculating the compression. Therefore, write operations on the column store do not directly modify compressed data of the so-called *main store*, but all changes go into a separate data structure called *differential buffer* as shown in Figure 4. Both structures are dictionary encoded, while the main store is further compressed using additional compression techniques. While the dictionary of the main store is a sorted array, which defines the mapping of a value to its value ID as the position of that value in the array, the dictionary of the differential buffer is an unsorted array, allowing for fast insertion of new values. The arbitrary order of values in the differential buffer dictionary slows down read operations, since a lookup of a value ID has complexity $O(N)$, or $O(logN)$ if an index structure, e.g., a B+/CSB+ tree is used for value ID lookup. A growing dif-

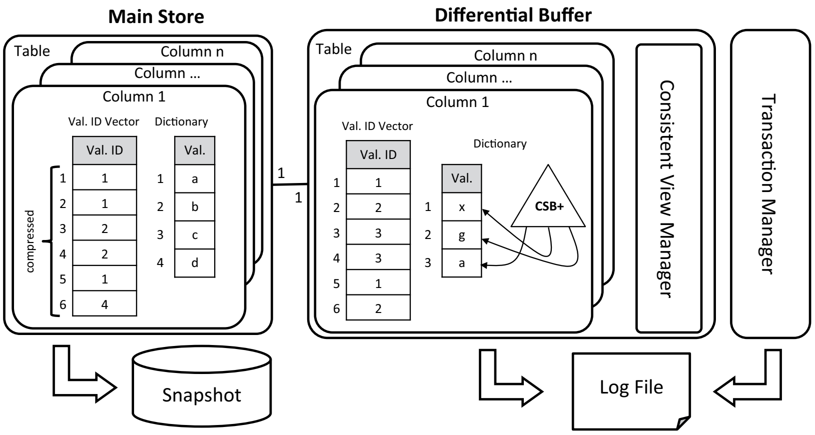


Figure 4: Operations on the Row Store and the Column Store

ferential buffer degrades read performance and increases memory usage since the value ID vector is not further compressed as in the main store and, in addition, the index of the dictionary, i.e., the CSB+ tree, grows fast if a column has many unique values. Therefore, differential buffer and main store are merged from time to time. The merge process is a reorganization of the column store integrating the differential buffer into the main store and clearing the differential buffer afterwards [KGT$^+$10].

4 Transaction Management

Enterprise applications require strong transactional guarantees. SanssouciDB uses a Multi Version Concurrency Control (MVCC) scheme to isolate concurrent transactions, while a physiological logging and snapshotting approach is used to persist transactional changes on non-volatile memory to provide fault tolerance. In the following, we give an overview of how the MVCC scheme exploits versioning of tuples provided by the insert-only approach and describe the logging scheme of SanssouciDB.

4.1 Concurrency Control

Isolation of concurrent transactions is enforced by a central transaction manager maintaining information about all write transactions and the *consistent view manager* deciding on visibility of records per table. A so-called *transaction token* is generated by the transaction manager for each transaction and encodes what transactions are open and committed at the point in time the transaction starts. This transaction token is passed to the consistent view manager of each table accessed and is used to emulate the same record visibility as

TID	TA State	CID
...	...	...
6	aborted	-
7	committed	7
8	open	-
9	committed	9
10	committed	8
11	committed	10

Table 1: Example for transaction information maintained by the transaction manager.

TID	New Rows	Invalidated Rows
< 8	1, 2	M:20, M:10, M:5
9	3	
10	4	D:3
11		D:2

Table 2: Consistent view information for transactions of Figure 3

at transaction start time.

For token generation, the transaction manager keeps track of the following information for all write transaction: (i) unique transaction IDs (TID), (ii) the state of each transaction, i.e., *open*, *aborted*, or *committed*, and (iii) once the transaction is committed, a commit ID (CID). While CIDs define the commit order, TIDs reflect the start order of transactions. This is exemplified in Table 1. From this information, the transaction manager generates the following values of the transaction token:

- *maxCID*: the highest CID, in our example maxCID=10
- *minWriteTID* and *closedTIDs*: together these values describe all transactions that are closed. In our example we have minWriteTID=7 and closedTIDs={9,10,11}. Changes made by a transaction T$_i$ with TID$_i$ $<$ minWriteTID or TID$_i$ $\in$ closedTIDs must be visible for transactions T$_j$ if TID$_j$ $\geq$ TID$_i$.
- *TID*: for a write transaction this is the unique TID of the transaction. For read transactions this is the TID that would be assigned to the next starting write transaction. Hence, different read transactions might have the same TID. A read transaction T$_i$ is not allowed to see changes made by a transaction T$_j$, if TID$_j$ $\geq$ TID$_i$. If a read transaction is promoted to a write transaction it may be necessary to update the transaction token with a new TID, because the original value might be used by another transaction.

All write operations insert a new record into the column store as part of the insert-only concept (c.f. Section 3.1). Rows inserted into the differential store by an open transaction are not visible to any concurrent transaction. New and invalidated rows are announced to the consistent view manager as soon as the transaction commits. The consistent view manager keeps track of all added and invalidated rows to determine the visibility of records for a transaction.

For every transaction T$_i$, the consistent view manager maintains two lists: one list with the rows added by T$_i$ and a second list of row IDs invalidated by T$_i$. For a transaction T$_j$, changes made by transactions with a TID smaller than T$_j$'s TID are visible to T$_j$. For a compact representation of change information, the consistent view manager consolidates the added row list and invalidated row lists of all transactions with a TID smaller than the *MinReadTID* into a single new row and a single invalidated row list. MinReadTID is defined as the maximum TID for which all changes written with the same or a lower TID may be shown to all active transactions. For all transactions T_i with a TID larger than MinReadTID, individual change information must be kept. Table 2 depicts the lists maintained by the consistent view manager for our example transactions of Figure 3: TA_1 inserts row 3, while TA_2 invalidates row 3 and adds row 4. The first row of Table 2 shows the consolidation of new and invalidated row lists for all transactions with TIDs smaller than MinReadTID=8.

To determine the visibility of tuples for transaction T_i, i.e., with TID_i=12, the consistent view manager interprets the transaction token as follows: (i) since for a running transaction T_i the condition $TID_i >$ MinReadTID holds, all changes listed in the consolidated list (first row in Table 2) are visible for T_i. In addition, all changes made by transactions T_j with $TID_j \leq TID_i$ are visible for T_i. In our example, this are the changes of transactions with TIDs 9, 10, and 11. All changes of T_k with $TID_i \leq TID_k$ are not visible for T_i.

While read operations can access all visible rows without acquiring locks, write transactions are serialized by locks on row level. Besides main store and differential buffer, which store the most recent version of a tuple, each table has a history store containing previous versions of tuples. The history store uses the CID for distinguishing multiple old versions of a tuple. In doing so, SanssouciDB provides a time-travel feature similar to the one described in [Sto87].

4.2 Logging and Recovery

Enterprise applications are required to be resilient to failures. Fault tolerance in a database system refers to its ability to recover from a failure and is achieved by executing a recovery protocol when restarting a crashed database, thereby restoring its latest consistent state before the failure. This state must be derived using log data that survived the failure, in the form of logs residing on a non-volatile medium. Writing log information to non-volatile memory is a potential bottleneck because it is bound by disk throughput. To prevent log writes from delaying transaction processing, SanssouciDB uses a parallel logging scheme to leverage the throughput of multiple physical log volumes.

To recover a table, the main as well as the differential store must be rebuilt in a consistent fashion. The main part of a table is snapshot to a non-volatile medium when main and differential store are merged (c.f. Section 3.1). In order to fall-back to a consistent state during recovery, redo information for all write operations since the last merge is logged to the *delta log*. The central transaction manager writes a *commit log* file recording the state transitions of all write transactions.

At restart of a crashed database, the main part is recovered using its latest snapshot and the differential buffer is recovered by replaying the delta and commit logs. While recovering the main store from a snapshot is fast, for example when using memory-mapped files on an SSD, replaying log information is a potential bottleneck for fast recovery.

As described in Section 3.1, the main store of a table as well as the differential buffer consists of a vector holding value IDs and a dictionary defining the mapping of value IDs to values. During normal operations, the dictionary builds up over time, i.e., a new mapping entry is created in the dictionary each time a new unique value is inserted. Both value ID vector and dictionary, must be rebuilt during recovery using the delta log.

To allow parallel recovery, we use a physiological logging scheme [GR93] that stores the insert position (row ID) and the value ID for each write operation, called a *value log*. The logging scheme also includes logs for the dictionary mapping (dictionary logs). A value log entry contains the TID, the affected attribute, the row ID, and the value ID of the inserted value defined by the dictionary mapping. A dictionary log contains the attribute, the value, and the value ID.

The system attributes of each record (TID and invalidated row) are logged in the value log format. Combined with commit logs written by the transaction manager, they are used to rebuild the added row and invalidated row lists of the transaction manager at recovery time. A commit log entry contains the TID, the transaction state, and a commit ID if the state is committed.

To reduce the number of log entries for a row, only value logs for attributes that were actually changed by an update operation are persisted. During recovery, the missing attribute values of a record, i.e., attribute values that are not updated but have the same value as the previous version of the record can be derived from the previous version of the record.

Dictionary entries are visible to other transactions before the writing transaction has committed, to prevent the dictionary from becoming a bottleneck during transaction processing. Therefore, dictionary entries must be managed outside the transactional context of a running transaction to prevent removal of a dictionary entry in case of a rollback. For example, if transaction T$_i$ inserts a dictionary mapping 'abc' $\rightarrow 1$ and is aborted after another transaction T$_j$ reads and uses the mapping for 'abc' from the dictionary, removing the mapping from the dictionary during rollback of T$_i$ would affect T$_j$, which must not happen. Logging dictionary changes outside the transactional context causes the problem that in case transactions are aborted a dictionary might contain unused values. However, unused entries are removed with the next merge of differential and main store and will not find their way into the new main store.

To speed up recovery of the differential buffer further, the value ID vector as well as the dictionary can be snapshot from time to time, which allows for truncating the delta log.

5 Parallel Aggregation and Join

SanssouciDB runs on a blade architecture with multiple cores per blade (see Section 2). The system can parallelize algorithms along two dimensions: across blades and within a blade. In the first case, we assume a shared-nothing architecture. Each blade is responsible for a certain partition of the data. In the second case, compute threads that run in parallel

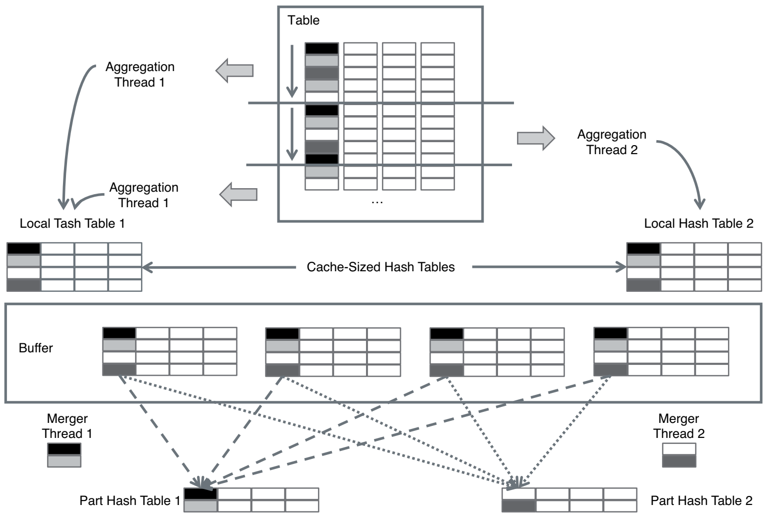


Figure 5: Parallel Aggregation

on one blade can access the same data on that blade, which is typical for a shared-memory architecture. In this section, we want to discuss aggregation and join algorithms developed for SanssouciDB.

5.1 Parallel Aggregation

Aggregation algorithms have been proposed for shared-nothing architectures and for shared-memory architectures. For a discussion on distributed aggregation across blades, we refer the reader to [TLRG08]. Here, we want to discuss the shared-memory variant is implemented utilizing multiple concurrently running threads. We first consider how the input data is accessed (see upper part of Figure 5). Let us assume, we can run n threads in parallel. Initially, we run n aggregation threads. Each aggregation thread (1) fetches a certain rather small partition of the input relation, (2) aggregates this partition, and (3) returns to step (1) until the complete input relation is processed. This approach avoids the problem of unbalanced computation costs: If we would statically partition the table into n chunks, one per thread, computation costs could differ substantially per chunk and thread. Using smaller chunks and dynamically assigning them to threads evenly distributes the computation costs over all threads.

Each thread has a private, cache-sized hash table, into which it writes its aggregation results to. Because the hash table is of the size of the cache, the number of cache misses are reduced. If the number of entries in a hash table exceeds a threshold, for example, when 80 % of all entries are occupied, the aggregation thread initializes a new hash table and moves the old one into a shared buffer. When the aggregation threads are finished, the buffered hash tables have to be merged. This is accomplished by merger threads. The buffered tables are merged using range partitioning. Each merger thread is responsible for a certain range, indicated by a different shade of grey. The merger threads aggregate their partition into so-called part hash table. Part hash tables are also private to each thread. The partitioning criterion can be defined on the keys of the local hash tables. For example, all keys, whose hashed binary representation starts with an 11, belong to the same range and are assigned to a certain merger thread. The final result can be obtained by concatenating the part hash tables (since they contain disjoint partitions).

The algorithm can be improved by allowing merger threads to run when the input table has not been completely consumed by the aggregation threads. Aggregation phases and merge phases alternate. This allows us to restrict the number of local hash tables in the buffer, thus reducing the memory footprint of the algorithm. Note, a lock is required to synchronize the partitioning operations on the input table and on the hash tables in the buffer. These locks are, however, only held for a very short period of time, because partition ranges can be computed quickly.

5.2 Parallel Join

Similar to the aggregation algorithm, SanssouciDB can compute joins across blades and within one blade utilizing multiple threads. To compute a distributed join across blades, SanssouciDB applies the well-known semijoin method: Let us assume a join between tables R and S on the set of join columns A. Tables R and S are stored on different blades (1 and 2). First, projection $\pi_A(R)$ is calculated on Blade 1. The projection retains the set of join columns A and applies duplicate removal. The projected result is then sent to Blade 2, where the intermediate result $T = \pi_A(R) \rtimes S$ is computed. T is the set of tuples of S that have a match in $R \bowtie S$. Projection $\pi_A(T)$ is sent back to the node storing R to calculate $U = R \bowtie \pi_A(T)$. The final join result is obtained by calculating $U \bowtie T$.

In the distributed join computation, some operations run locally on a blade. For these operations, parallel algorithms tailored to shared-memory architectures can be applied. Because the computation of distinct values and aggregation are closely related, projection $\pi_A(R)$ on Blade 1 can be computed using a slightly modified version of the parallel aggregation algorithm above. Furthermore, the semijoins can also be computed locally in parallel. In the following, we want to sketch a parallel algorithm for join computation.

Just like aggregation, joins can be computed based on hash tables. The algorithm is depicted in Figure 6. The columns with the dashed lines contain row numbers. Row numbers are not physically stored in SanssouciDB. They are calculated on the fly when required. In the preparation phase depicted at the left hand side, the values of the smaller input relation's join attributes and their corresponding row numbers are inserted into a hash table. This insertion is executed in parallel by concurrently running join threads. Again, each thread has a private cache-sized hash table that is placed into the buffer when a certain threshold is reached. The buffer is occasionally merged into part hash tables. The result consists of a set of part hash tables with key-list pairs. The pair's key corresponds to a value from a join column, and the pair's list is the list of row numbers indicating the posi-

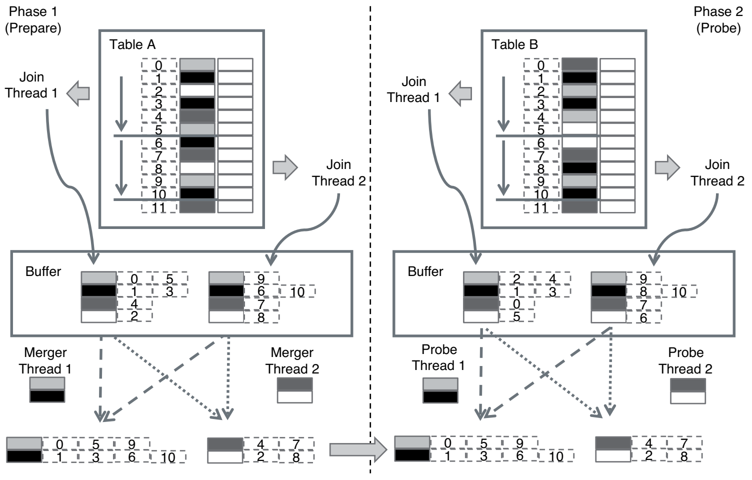


Figure 6: Parallel Join

tion of the value's occurrence in the input relation. Again, the keys in the part hash tables are disjoint.

In the second phase, the algorithm probes the join columns of the larger input relation against the part hash tables. Probing works as follows:

1. Private cache-sized hash tables for the larger input relation are created and populated by a number of concurrently running threads as described above. When a hash table reaches a certain threshold, it is placed in the buffer.

2. When the buffer is full, the join threads sleep and probing threads are notified. In Figure 6, we copied the resulting part hash tables from the preparation phase for simplicity. Each probing thread is responsible for a certain partition. Because the same partitioning criterion is used during the first phase of the algorithm, all join candidates are located in a certain part hash table. If the probing thread finds a certain value in a part hash table, it combines the rows and appends them to the output table.

3. When all hash tables in the buffer have been processed, the join threads are notified.

4. The algorithm terminates when the buffer is empty and the larger input relation has been consumed. The result is the joined table.

This algorithm can be improved by materializing the join result as late as possible. For example, the result of the join can be a virtual table, that contains references to the original tables by keeping a list of row pairs. Furthermore, the join algorithm has to be aware of the columnar data layout. This is also true for the aggregation algorithm. As described so far, when processing a partition, an aggregation or join thread reads values from the participating columns row by row. Row-wise access is expensive in a column-oriented database system, because row-wise access provokes many cache misses. To remedy this situation, we can use a special hash-table implementation that allows column-wise insertion. The details are however beyond the scope of this paper.

5.3 Business Analytics on Transactional Data

Good aggregation performance is one of the main objectives in data warehouses. One approach to achieve the necessary performance in disk-based relational data warehouses is to consolidate data into cubes, typically modeled as a star or snowflake schemas. In a cube, expensive joins and data transformations are pre-computed and aggregation is reduced to a mere scan across the central relational fact table. Due to the possibly large size of fact tables in real-world scenarios (a billion rows and more are not unusual), further optimizations such as materialized aggregates, became necessary.

With the introduced aggregation and join algorithms that exploit modern hardware, redundant data storage using cubes and materialized aggregates are not necessary anymore. Executing business analytics on transactional data that is stored in its original, normalized form becomes possible with SanssouciDB. Avoiding materialized aggregates and the need for the star schema dramatically reduces system complexity and TCO, since cube maintenance becomes obsolete. Calculations defined in the transformation part of the former ETL process are moved to query execution time. Figure 7 shows an example of an architecture that uses views that can be stacked in multiple layers. Any application on the presentation layer can access data from the virtually unified data store using views. Views store no data but only transformation descriptions such as computations, possibly retrieving data from several sources, which might again be views. Virtual cubes are similar to views but provide the same interface as real cubes.

Our architecture enables the instant availability of the entire data set for flexible reporting. Moreover, this approach fulfills many of the ideas mentioned in the context of BI 2.0 discussions, where BI is "becoming proactive, real-time, operational, integrated with business processes, and extends beyond the boundaries of the organization" [Rad07]. In the next section, we will showcase some sample applications that make use of our new architecture.

6 Application Examples

We have extensively used SanssouciDB in enterprise application prototypes to leverage its technological potential. In the following, we want to discuss our findings by showing how SanssouciDB can be used to implement the dunning and the availability-to-promise applications.

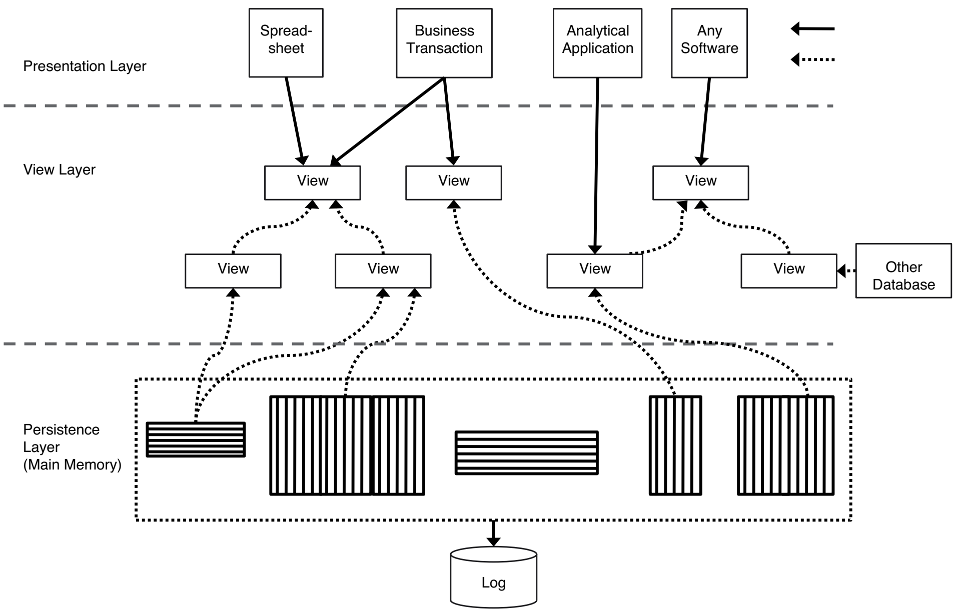


Figure 7: The View Layer Concept

6.1 Dunning

The dunning business process is one of the most cost-intensive business operations. Dunning computes the balance between incoming payments and open invoices for all customers. Depending on the amount and the delay of an outstanding payment, different dunning levels are assigned to customers. Depending on the dunning level, pre-defined actions are triggered, e.g., service blacklisting. Performing dunning runs in current enterprise systems has an impact on OLTP response times, because of the large number of invoices and customers accounts that must be scanned. To reduce the impact on the response time of the operational system, dunning runs are typically performed at the night or over tge weekend. Our field study at a German mobile phone operator revealed that dunning runs in current enterprise applications can only be performed at daily or weekly intervals for all customers.

The dunning process is mainly a scan of a long table containing millions of entries. In a prototype, we adapted dunning algorithms that were formerly implemented in the application layer and re-implemented them using stored procedures in our new storage engine. To join invoice data with customer data and to calculate the amounts due, our implementation makes heavy use of the join and aggregation algorithms introduced in Section 5. Our results show that SanssouciDB is able to improve the execution time of the dunning run from more than 20 minutes to less than one second.This outcome shows that in-memory technology is capable of improving the response time of existing applications by orders of magnitude.

As a consequence, the former batch-oriented dunning run could be transformed into an interactive application. For example, a check of the current dunning status over all customers can now be performed instantaneously. This enables managers to query the top ten overdue invoices and their amount on their personal cellular phone, notebook, or any mobile device with an Internet connection at any time.

6.2 Availability-To-Promise

ATP provides a checking mechanism to obtain feasible due dates for a customer order. This is done by comparing the quantities of products which are in stock or scheduled for production against the quantities of products which are assigned to already promised orders. A common technique in current Supply Chain Management systems is to use materialized aggregates to keep track of stock quantities, which results in having a separate aggregate for each product. This means that a new shipment would increase the value of each shipped product's aggregate, while the assignment of products to a confirmed customer order would decrease it. We analyzed an existing ATP implementation and found the following: Although the use of aggregates reduces the necessary amounts of I/O operations and CPU cycles for the single ATP check itself, it introduces the following disadvantages:

- *Redundant Data*: One problem that arises in association with materialized aggregates is the need to maintain them. To preserve a consistent view on the data across the whole system, every write operation has to be propagated to the materialized aggregates. Even if the updates are triggered immediately, they still imply delays causing temporary inconsistencies. Additionally, even if the amount of I/O operations and CPU cycles is reduced to a minimum for the check itself by using aggregates, the overall sum of required operations might be higher due to synchronization as well as maintenance and costly back calculation of the aggregates.
- *Exclusive Locking*: All modifications to an aggregate require exclusive access to the respective database entity and block concurrent read and write processes. The downside of locking is obvious, as it queues the incoming requests and affects the performance significantly in case of a highly parallel workload.
- *Inflexible Data Querying*: Materialized aggregates are tailored to a predefined set of queries. Unforeseeable operations referring to attributes that were not considered at design time cannot be answered with these pre-aggregated quantities. Those attributes include, for example, the shelf life, product quality, customer performance, and other random characteristics of products, orders, or customers. Additionally, due to the use of materialized aggregates, the temporal granularity of the check is fixed. Once the aggregates are defined and created, based on, for example, the available quantities per day, it is not possible to perform ATP checks on an hourly granularity.
- *Inflexible Data Schema Extensions*: The previously mentioned inflexibility of not being able to change the temporal granularity of a single check indicates another

related disadvantage: the inability to change the data schema, once an initial definition has been done. The change of the temporal check granularity or the inclusion of a previously unconsidered attribute is only possible with a cumbersome reorganization of the existing data.

- *No Data History*: Maintaining aggregates instead of recording all transactions enriched with information of interest means to lose track of how the aggregates have been modified. In other words, no history information is available for analytics or for rescheduling processes.

As we have outlined in Section 5.3, SanssouciDB can execute business analytics on operational data. No star schema or materialized aggregates are required. To leverage this advantage for our ATP implemenation, we developed a prototype that was designed with the following aspects in mind: Pre-aggregated totals were completely omitted by storing stock data at the finest granularity level in SanssouciDB. For an ATP check, this up-to-date list is scanned. Removing the materialized aggregates allowed us to implement a lock-free ATP check algorithm. Eliminating locks has the advantage of concurrent data processing, which works especially well in hotspot scenarios, when multiple users perform ATP checks concurrently.

For the prototype implementation, we used anonymized customer data from a Fortune 500 company with 64 million line items per year and 300 ATP checks per minute. Analysis of the customer data showed that more than 80 % of all ATP checks touch no more than 30 % of all products. A response time of 0.7 s per check was achieved using serial checking. In contrast, we were able to execute ten concurrently running checks with our adapted reservation algorithm in a response time of less than 0.02 s per check.

Our approach eliminates the need for aggregate tables, which reduces the total storage demands and the number of inserts and updates required to keep the totals up to date. Because insert load in SanssouciDB could be reduced, the spare capacity can be used by queries and by instant aggregations. Since all object instances are traversed for aggregation, fine-grained checks on any product attribute are possible. The old ATP implementation was kept lean by focusing on a minimum number of attributes per product. For example, printers of a certain model with a German and an English cable set are grouped together or both cable sets are added to improve performance. With the help of SanssouciDB, fine product attributes could be managed in a fine-grained manner. Thus, printers including different cable sets could be considered as individual products during ATP checks.

7 Conclusion

With SanssouciDB, we have presented a concept that we believe is the ideal in-memory data management engine for the next generation of real-time enterprise applications. Its technological components have been prototypically implemented and tested individually in our research. To conclude this paper, we present a few cases where the application of the concepts of SanssouciDB has already led to promosing results.

SAP's latest solution for analytical in-memory data management called HANA (High-Performance Analytical Appliance) uses many of the concepts of SanssouciDB, and is currently being rolled out to pilot customers in different industries. HANA's performance impact on common reporting scenarios is stunning. For example, the query execution times on 33 million customer records of a large financial service provider dropped from 45 minutes on a traditional DBMS to three seconds on HANA. The speed increase fundamentally changes the company's opportunities for customer relationship management, promotion planning, and cross selling. Where once the traditional data warehouse infrastructure has been set up and managed by the IT department to pre-aggregate customer data on a monthly basis, HANA now enables end users to run live reports directly against the operational data and to receive the results in seconds.

In a similar use case, a large vendor in the construction industry is using HANA to analyze its nine million customer records and to create contact listings for specific regions, sales organizations, and branches. Customer contact listing is currently an IT process that may take two to three days to complete. A request must be sent to the IT department who must plan a background job that may take 30 minutes and the results must be returned to the requestor. With HANA, sales people can directly query the live system and create customer listings in any format they wish, in less than 10 s.

A global producer of consumer goods was facing limitations with its current analytical system in that it was not able to have brand and customer drilldowns in the same report, which is no possible since joins are no longer pre-computed. Finally, the query execution times for a profitability analysis application of the same customer were reduced from initially ten minutes to less than ten seconds.

References

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented Database Systems. *PVLDB*, 2(2):1664–1665, 2009.

[ADH02] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.

[GKP$^+$11] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE - A Hybrid Main Memory Storage Engine. *PVLDB*, 2011.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[KGT$^+$10] Jens Krüger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Database Systems for Advanced Applications, 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II*, pages 291–305, 2010.

[LSF09] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In *Datenbanksysteme in Business, Technologie und Web (BTW 2009), 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 2.-6. März 2009, Münster, Germany*, pages 486–497, 2009.

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1–2, 2009.

[Rad07] N. Raden. Business Intelligence 2.0: Simpler, More Accessible, Inevitable, 2007. `http://intelligent-enterprise.com`. Retrieved January 14th 2011.

[RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89, 1999.

[RR00] Jun Rao and Kenneth A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486, 2000.

[SEJ$^+$11] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting In-Memory Database Performance for Automating Cluster Management Tasks. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16 2011, Hannover, Germany*. IEEE, Apr 2011. (to appear).

[SMA$^+$07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160, 2007.

[Sto87] Michael Stonebraker. The Design of the POSTGRES Storage System. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289–300, 1987.

[TLRG08] David Taniar, Clement H. C. Leung, J. Wenny Rahayu, and Sushant Goel. *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons, 2008.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55–67, 2000.

[WPB$^+$09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.