Efficient recovery in a data-flow based storage system

Martin Ek

November 8, 2017

Abstract

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Soup

Modern web applications face an increasingly difficult performance problem in relation to their storage needs: traditional relational databases perform too poorly when used in isolation. Either by failing to scale to a large amount of concurrent users, or by taking too long to return results for more expensive aggregation queries.

This is often worked around by building up intricate cache hierarchies [15], similar to the multi-level cache hierarchy readers might be accustomed to seeing in their processors. This solves both performance problems: concurrent readers can access materialized data from the upper levels of the cache hierarchy, achieving low latency and high throughput even when faced with large amounts of clients. Expensive queries then only need to be performed once, as long as the result is invalidated when the underlying data changes. Write operations still need to change the underlying storage layer however, but are now also responsible for clearing out or updating the materialized cache levels.

This adds an extra layer of complexity, and is a trade-off accepted by almost every modern day developer that faces more than a trivial amount of load to their storage system.

Soup sets out to solve this by providing a single high performant system, removing the need for manual materialization altogether. The current prototype scales to millions of writes and reads per second [2], and works both on a single machine and in a distributed setting.

Soup does this by materializing data automatically in a data-flow graph, building on existing streaming data-flow research [14, 12] by combining it with ideas from performant materialized view solutions [8, 7].

2.1.1 Data-flow Graph

Soup turns base table schemas and a pre-defined list of queries into a data-flow graph that doubles as a set of materialized views. The base tables form the root nodes of the graph, and all writes propagate from here. The graph itself resembles the query graph of a traditional relational database system [4], with the graph's intermediate nodes being relational operators.

However, whereas a relational database management system's query graph is used primarily to retrieve data, by executing operators to fetch data from durable storage, Soup's graph does the exact opposite. The graph is defined ahead of queries being received, and writes stream through the relevant nodes in the graph - starting from the base nodes. This skews the majority of the system's work towards writes, as results are materialized at different nodes throughout the graph.

The query graph includes at least one materialized node per pre-defined query, located at the bottom of the graph as leaf nodes. This means that reading from the system is a matter of finding the relevant leaf node, and retrieving data from the node's materialized storage. This is one of the elements that make Soup resemble more of a key-vaule store than a traditional RDBMS. Even though Soup supports advanced SQL queries, both mutations and retrievals are done with a key, which then maps to a corresponding value.

On an elementary level one could see Soup creating a separate graph for each query, similar to query graphs in traditional database systems, but that would have been quite inefficient. For one it would have lead to Soup materializing a lot of duplicate data for intermediary materialized nodes, and base table writes would have had to be propagated to more nodes than necessary. Migrations would also have taken considerably longer time, as adding queries would mean constructing a completely new query graph. Instead Soup uses multi-query optimization techniques to create common sub-graphs of nodes that can be used in multiple queries.

2.1.2 Materialization

Similar to in traditional relational databases, writes are inserted into the table directly - a base node in Soup. Whereas this marks the end of the operation for

an RDBMS, Soup will continue by propagating the write throughout the query graph, storing the result at materialized nodes.

The insertions are handled with a group-commit protocol, where records are batched up and finally inserted into a durable log before being propagated to the base node.

Soup primarily materializes leaf nodes in the graph, to enable fast read queries. In some cases intermediary operators might also choose to materialize the records they receive before they forward them through to their child nodes. This is especially the case for stateful operators, such as extremum queries, as these need to incrementally update their result - instead of having to go through and re-calculate the state on every new record.

Nodes can also be partially materialized, which lets each node choose which rows to keep in their internal state. This resembles a regular caching system, where keys can be evicted after a period of inactivity.

2.1.3 Migrations

Relational databases require developers to pre-define a set of table schemas for their application. Changes to these throughout a project's lifecycle are often inevitable, however the process of performing migrations in a live system without downtime is usually far from trivial [17]. This is one of the core issues Soup sets out to solve, by providing a database system that lets developers change both the base table schemas and the pre-defined queries at any point.

Soup constructs a new graph for added queries. This is done by finding overlaps with the existing query graph nodes, to avoid duplicate materializations and to reduce the amount of work done by each migration. Soup marks nodes as partially materialized when possible, which lets the system start processing requests for the newly defined query right away, by only fetching each key from ancestor nodes when necessary. For fully materialized nodes, Soup will replay all relevant base table writes from the closest materialized node. This delays the point where Soup can start processing requests, but has the advantage of not delaying subsequent requests to retrieve materializated keys later on.

Modifications to base table schemas are done in-place, by having Soup's base nodes internally include the full history of columns for that specific table.

2.1.4 Transactions

An increasing amount of real-world database system users have found relaxed consistency guarantees to be sufficient in the last few years, as it greatly simpli-

fies the problem of replicating large amounts of data across data centers with high performance [10]. There are nonetheless without doubt a wide variety of usecases for transactions, where the eventual consistency that Soup offers is insufficient. To cater to these types of applications, Soup allows developers to opt-in for transaction support, by defining their base nodes as transactional.

Soup's transaction support is carefully designed to avoid slowing down non-transactional workloads. Using optimistic concurrency control, Soup returns timestamped tokens for reads, which can later be used to verify the validity of the data that was read. Writes can only be performed at the end of a transaction, when clients request their transactions to be committed. The timestamp part of the token helps Soup validate if concurrent clients interferred with the data, which lets the system notify the client by aborting the ongoing transaction.

2.2 Rust

Rust [1] is an open-source systems programming language that guarantees memory safety while maintaining a minimal runtime. Rust is sponsored by Mozilla, where it is used to develop Servo - a completely new browser engine.

Listing 2.1: "Hello World in Rust"

```
fn main() {
  println!("Hello World!");
}
```

When choosing a language, developers are often forced to compromise between higher level abstractions and performance. Large and latency sensitivy projects like databases often opt for the latter through low-level languages like C, which avoid expensive runtime safety checks. Rust removes this dilemma altogether by providing developers with both the fine-tuned control and performance they are used to in low-level languages, while offering abstractions developers might be familiar with from interepretted languages.

One of Rust's key features is providing compile time safety both in terms of types and memory. The latter is done through an ownership model which lets developers program mostly without thinking about memory allocation and deallocation, without the lowered performance of using something like a garbage collector. Each variable in Rust is assigned one and only one owner, and the variable is deallocated - or dropped - when that owner goes out of scope.

Chapter 3

Recovery

3.1 Traditional Recovery

At the core of most database systems you will find a log manager, which maintains a sequence of log records on disk. Logging is essential in maintaining fault tolerancy and durability, and is used for a variety of issues: ensuring the durability of committed data, recovering from failures, and rolling back aborted transactions.

Even to this day one of the most popular algorithms for recovery is ARIES [13] (Algorithm for Recovery and Isolation Exploitation Semantic), which uses a write-ahead log to make sure that a persistent record of changes is maintained before the actual changes themselves. Upon recovery ARIES makes sure that all actions from the log prior to a crash are repeated (*REDO*), then reverts the operations belonging to failing transactions (*UNDO*). The former makes sure that durability is respected, while the latter maintains atomicity.

Whereas the logging protocol used for recovery varies from implementation to implementation, the principle behind a write-ahead log remains the same. By maintaining a log of actions *before* the changes themselves are made durable, we make sure that we can recover from a potential crash. For transactional systems we also maintain enough data to be able to achieve atomicity for transactions.

3.2 Logging in Soup

Soup only logs entries from committed transactions, greatly simplifying the recovery protocol. The log is written prior to entries being forwarded through the query graph, which makes sure that all changes are persisted to disk before being made accessible to the client. Soup does however make use of a *group-commit scheme* [5], buffering up a series of packets before finally incurring an I/O write to persist the group as one unit.

At the time of writing, Soup defaults to gathering up 256 operations before flushing the group to disk, drastically increasing the throughput of the system. Not only does this reduce the amount of I/O operations per request, merging the data from multiple requests into a single group also means less packets to feed through the Soup query graph. The log entries themselves are JSON [6] serialized arrays of Soup's data types.

Recovery in Soup is then a matter of replaying the writes from the persisted log on start-up, feeding them into the query graph's base nodes as if they were standard mutations. Similar to the group-commit protocol used for logging, the log entries are batched up into larger groups on recovery, to reduce the amounts of packets flowing through the Soup query graph.

3.3 Recovery in Main-Memory Databases

Accessing and writing to durable storage is consistently a bottleneck in database storage systems. Even though database systems that rely on non-volatile storage for durability can have large amounts of data available in RAM, writes still have to be persisted to disk ahead of being made available to clients. With the rise of hardware that is capable of storing entire applications' datasets in their main memory, this is often seen as inefficient, and has lead to an increase in so called main memory database systems.

As these main memory database systems rely on fast volatile storage as their main data store, traditional techniques for maintaining ACID principles seem unappealing. Almost all storage systems want to maintain some measure of durability nonetheless however, which means that state needs to persisted to durable storage at some point.

Taking periodic checkpoints of the entire system's state is one way of avoiding total data loss in the event of a crash. Checkpointing used alone only achieves partial durability however, as it is unfeasible to take a checkpoint after every operation. Because of this some main memory systems also rely on so-called

Listing 3.1: "Recovering from the Soup log"

```
BufReader::new(file)
    .lines()
    .filter_map(|line| {
        let line = line.unwrap();
        let entries: Result<Vec<Records>, _> = serde_json::
            from_str(&line);
        entries.ok()
    })
    .flat_map(|r| r)
    // Merge packets into batches of RECOVERY_BATCH_SIZE:
    .chunks(RECOVERY_BATCH_SIZE)
    .into_iter()
    .map(|chunk| chunk.fold(Records::default(), |mut acc,
        ref mut data| {
        acc.append(data);
        acc
    }))
    // Then create Packet objects from the data:
    .map(|data| {
        let link = Link::new(local_addr, local_addr);
        if is_transactional {
            let (ts, prevs) = checktable.recover(global_addr
                ).unwrap();
            Packet::Transaction {
                link, data, tracer: None,
                state: TransactionState::Committed(ts,
                    global_addr, prevs)
            }
        } else {
            Packet::Message { link, data, tracer: None }
        }
    })
    .for_each(|packet| self.handle(box packet));
```

K-safety, where data is replicated to K+1 replicas [16], ensuring that the system could survive up to K failing nodes and still maintain full durability. K-safety is because of this naturally sensitive to total failures, where for example entire data centres fail.

A third alternative is maintaining a log of actions, which in VoltDB is referred to as command logging [11]. Whereas a fine-grained system like ARIES would log each individual modification, command logging would stick to only persisting the actual SQL query, which could then later be re-executed. This results in a significantly lower volume being logged, and a simpler recovery process after failures. This is akin to the recovery system currently being used in Soup, where each write request is logged with its corresponding data.

With only a coarse-grained command log recovery, times would steadily increase as an application gains more data, as all log entries would have to be processed to reach the system's previous state. This is unfeasible, and systems like VoltDB opt for checkpointing as a way of minimizing the size of the log that has to be replayed during recovery.

3.4 Snapshotting

Snapshotting a running Soup instance's entire state is far from trivial. Main memory systems like VoltDB leverage checkpointing by persisting the transactional state of commmitted transactions, using log sequence numbers to be able to track which updates have been reflected on disk. In Soup, state is materialized at a variety of nodes throughout the query graph, and updates have no timestamps or sequence numbers attached to them. Updates also propagate through the graph asynchronously, and a specific update is likely to reach different points in the graph at separate times. Taking a global snapshot of the entire graph would thus mean capturing nodes at different logical points, as a write might be in the process of propagating throughout the graph.

The nature of Soup's update propagation through a query graph resembles the communication done in a distributed system, where access to a common clock is rare. Being able to observe the global state in a distributed system is an immensely useful property, and is crucial to resolving problems such as deadlock detection.

Chandy and Lamport first introduced the problem of a distributed snapshot in "Distributed Snapshots: Determining Global States of Distributed Systems" [3], which has since been the inspiration for a wide variety of work within the field. Chandy and Lamport presented a solution to be used in a distributed

system with preserved message ordering, using first-in first-out channels. They resolved two main issues: deciding when to take a snapshot, and selecting which messages should be a part of said snapshot.

The key insight in [3] is to introduce a marker message, that can be used as a separator between messages that should be included in a snapshot, and messages that should not. Processes that receive a snapshot marker should immediately take a snapshot of all messages received prior to the marker, and forward the resulting state to a process capable of assembling all local snapshots to a global view of the system. Because of the channels' FIFO property, snapshots will not include messages that arrive after a marker at a node. This results in a requirement of $O(e)$ messages to initiate a snapshot, where e signifies the amount of edges in the graph. Since these messages can be sent out in parallel, the time to complete a snapshot is $O(d)$, where d is the diameter of the graph.

Lai and Yang [9] later extend this scheme with support for non-FIFO channels, and obleviate the need for explicit control messages by piggy-backing the required information onto existing messages.

3.4.1 Snapshotting in Soup

Soup's current recovery technique involves going through all log entries available, and processing each individual entry as if it was a new write to the system. No matter the recovery performance, the size of the log will regardless continue to grow with the lifetime of each Soup application. By introducing snapshotting of materialized nodes, Soup would be able to discard a significant part of the log at each snapshot, avoiding a continual growth in log size.

Soup's query graph communicates over FIFO channels, which makes it possible to rely on Chandy-Lamport's marker technique to signify which updates should be considered a part of the snapshot. Implemented naively this would lead to the entire graph snapshotting at the same time however, which would undoubtely cause a slow down in throughput. Thus we can say that we want snapshots to happen at *logically* the same time, but not at the same *physical* instant.

This adds a layer of complexity. Soup's materialized state is kept in a lock-free hash map, and there is no log kept at each separate node. This means that a node's internal state has to be cloned when a snapshot request is received, and that no additional updates can be processed until that has happened. Doing so would however mean either snapshotting at roughly the same physical time across the graph, or blocking additional updates for a period of time.

The solution this thesis proposes is to maintain a buffered log of writes that

are received after the snapshot marker, letting the system re-create the state at the point the snapshot marker was received. If the system's state at that point is denoted as S_n, and $L_{n..m}$ signifies the updates received from n to m, S_n can be re-created through $S_m - L_{n..m}$. This lets individual nodes delay initializing the snapshot process, preventing a global performance hit to the system.

When a node in the graph eventually snapshots its state, the base node should be notified of its completion. The snapshots are already persisted to disk, so including the actual data in these messages are unneccessary. They should on the other hand should include an identifier to the current snapshot request, so that the system eventually knows when an entire graph has performed the same snapshot. At that point the identifier of the snapshot can be persisted to disk, while simultaneously discarding the log prior to the snapshot.

Recovery is then a matter of first loading each materialized node's state from their local snapshot, then replaying the rest of the log entries available.

Chapter 4

Results

Chapter 5

Discussion

Bibliography

[1] The rust programming language. [Online; accessed 7-November-2017].

[2] Unpublished soup paper (attached).

[3] Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.

[4] Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.

[5] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. *Implementation techniques for main memory database systems*, vol. 14. ACM, 1984.

[6] The JSON data interchange format. Tech. Rep. Standard ECMA-404 1st Edition / October 2013, ECMA, Oct. 2013.

[7] Kate, B., Kohler, E., Kester, M. S., Narula, N., Mao, Y., and Morris, R. Easy freshness with pequod cache joins. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 415–428.

[8] Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., and Shaikhha, A. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal 23*, 2 (Apr 2014), 253–278.

[9] Lai, T. H., and Yang, T. H. On distributed snapshots. *Inf. Process. Lett. 25*, 3 (May 1987), 153–158.

[10] Lu, H., Veeraraghavan, K., Ajoux, P., Hunt, J., Song, Y. J., Tobagus, W., Kumar, S., and Lloyd, W. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 295–310.

[11] Malviya, N., Weisberg, A., Madden, S., and Stonebraker, M. Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on* (2014), IEEE, pp. 604–615.

[12] McSherry, F., Murray, D. G., Isaacs, R., and Isard, M. Differnetial dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)* (Asilomar, California, 2013).

[13] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[14] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[15] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[16] Ren, K., Diamond, T., Abadi, D. J., and Thomson, A. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1539–1551.

[17] Xu, J. Online migrations at scale, Feb. 2017. [Online; accessed 4-November-2017].