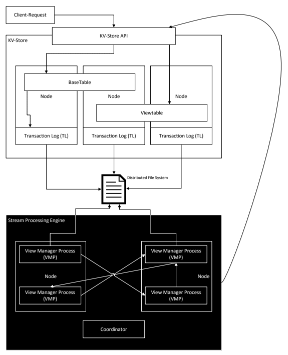
# View Maintenance Event Processing - Architectural Design

Tobias Schwan

September 8, 2015

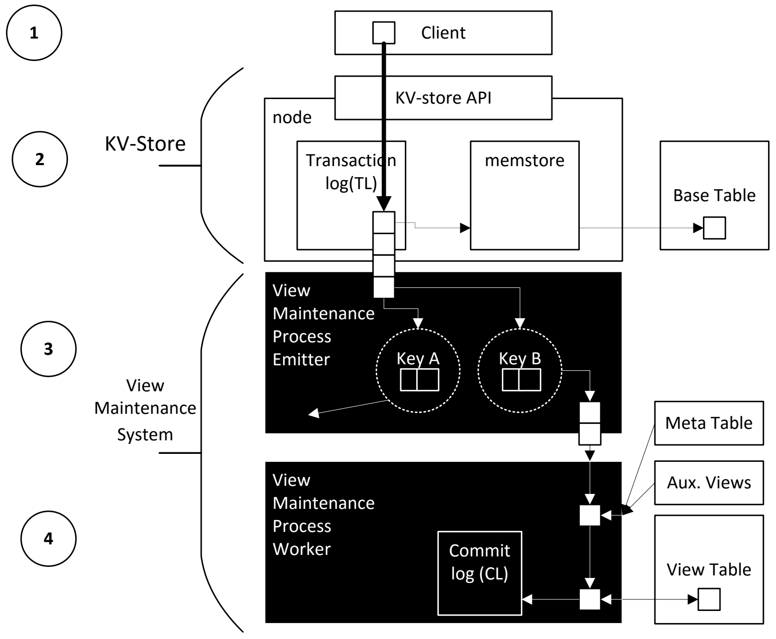
## 1 View Maintenance System

This excerpt describes the design of our View Maintenance System based on event processing (VMSev) . We present main architectural components and lay out structural differences to the original View Maintenance System (VMS) by Adler (2015). Both systems integrate with a typical KV-store. The general system design can be gathered from figure 1. There we see two major components. First illustrated in white, a traditional KV-store setup with a Transaction Log (TL) as persistence and recovery layer. Second, our event or stream processing engine. Both components can be distributed among several nodes connected by a distributed file system. This file system connects KV-nodes on the micro and VMSev components on the macro level. Nodes from our Stream Processing engine connect directly with one anonther. Client as well as view maintenance requests (put, get, delete) are handled by the KV-store API. While client requests are performed on the base table, the VMSev mirrors those on the view table. Like the original system we provide the ability to separate the maintenance engine from the KV Store. It is the preferred running mode over a KV node extension to avoid load-distribution scenarios in which it reduces the performance of the KV store. The underlying distributed file system has proven to withstand the additional load. Figure 1 hence shows our two components connected via the distributed file system.



The VMSev is represented in black. It is designed as a self-contained, fast failure, fault-tolerant system where load is distributed within. Every node hosts several general purpose View Manger Processes (VMP). According to the present load, the VMPs act in two major fashions. They either emit or work with data. In the following data is considered as tuples consisting of the row key and value. Usually the working phase consists of more than one step resulting in additional VMP. All processes are connected with one another. This is explained in more detail in an upcoming section. Load is balanced by the coordinator, which in our scenario mainly means the assignment and spawn of different VMP roles. Our system is attributed as fast failure since VMPs end themselves in error scenarios to free up space so new VMPs can be spawned. In the following we demonstrate that this approach does not threaten our consistency.

Figure 2 shows the inner workings of the VMS. A client request is passed on to the base table via the KV-store and to the view table via VMS. As described in previous work, we follow an *incremental* and *deferred* view maintenance strategy. It is motivated by the distributed and asynchronous nature of today's KV-stores. The approach is *incremental* since only those view table rows are updated where a base table operation exists. It is *deferred* since we separate client and base table from VMS and view table in order to reduce client update latency and maintain scalability.

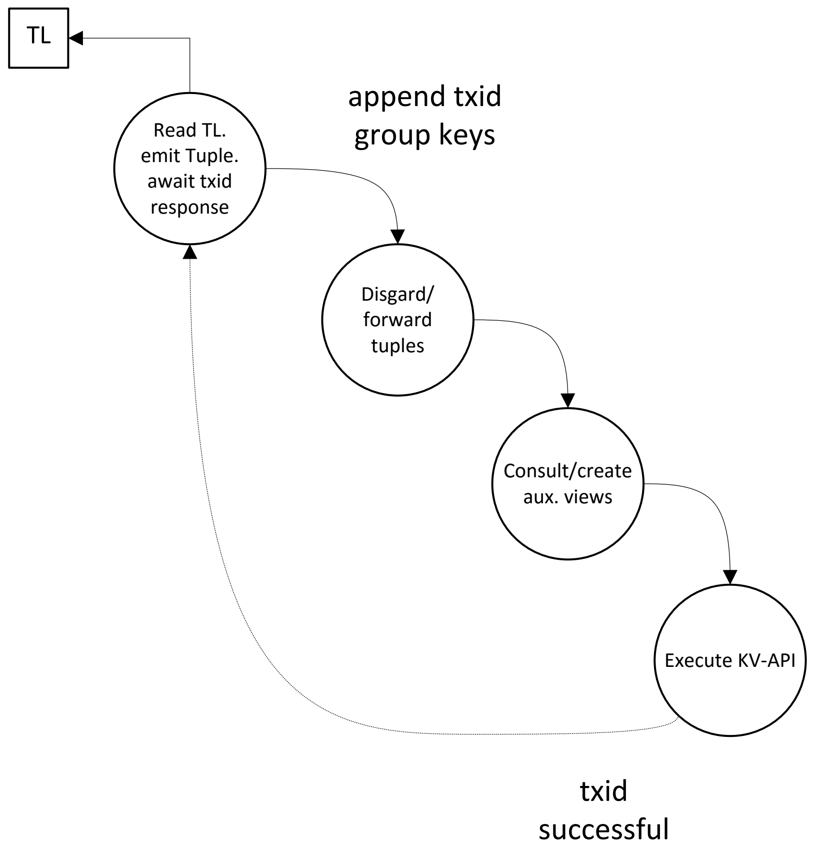


*Step 1 (client request):* via the KV-API a client request arrives at the KV-store. It is routed to the node which servers the key range. There the operation is handled.

*Step 2 (TL insert):* every client operation is written to the transaction log by the appropriate node. In case of a successful write the operation is applied to the memstore.

*Step 3 (partitioning):* over the distributed file system an VMP emitter fetches the client operation from the TL. With fetching operations form the TL instead of intercepting or even calling the KV-API directly we follow the work from Adler (2015). There four main reason are stated. (i) the sequential order of requests is maintained (ii) due to the asynchronous reading of the TL no additional load is imposed on the KV-store (iii) the persistence TL provided a recovery option for both the KV as well as the VMS (iv) elementary operations reduce complexity for incremental and deferred processing. After read from the TL operations are then grouped by the unique row key and forwarded to VMP workers. Hence operations on the same key are only processed by one VMP. Furthermore we provide an transaction identifier (txid) to assure sequential processing of requests on the same table row. A detailed analysis on processing semantics will be described in the following section.

*Step 4 (view update):* based on the forwarded tuple a VMP worker takes over, calculates possible auxiliary views and processes the client request to the view table via the KV-API. Auxiliary views as well as view table definitions are either held within the VMS or persisted via the distributed file system. Furthermore it is possible that auxiliary views are expressed as additional columns in the view table. Their role will be revealed in a later section. The shown commit log should be considered optional as the crash of VMP is covered in our architectural design through different mechanisms covered next. Using a Commit log makes only sense to recover of an total system failure or cluster shutdown.



Within our VMS four major processing steps exist for every view type. Figure 3 shows them and their order. It is not necessary for them to be executed on different nodes or even different processes. Furthermore all shown elements can exist one or n times. Our VMS scales with the number of processes and nodes while preserving the shown order. To maintain the sequential order of client request on row keys we introduce two mechanisms. i) First a unique and sequential transaction id (txid) is emitted with every tuple by the VMP that reads the transaction log. The txid can either be generated or passed one from the TL, since each operation there already has such an identifier. The latter can aid recovery operations. Then tuples are grouped according to their row key and further processed. ii) Next an evaluation of the tuples takes place in order to apply a condition expression, for example a selection or join condition. In our current system even operations enter the VMS that might not be relevant for the given view type. In the analysis part we clarify the impact on performance by comparing the layout shown here with one where the evaluation takes place before emitting tuples to the remaining VMS. When tuples have passed the last step possible auxiliary views are calculated and maintained. Based on that the actual view table is maintained. iii) Our second mechanism to avoid disorganized processing is the use of historical information of every row value. It is considered our third step. Adler (2015) introduced the concept of delta tables where the previous row value is maintained alongside the present value. With these two metrics, *txid* and *additional data*, we can clearly say whether a tuple has been processed or not and whether it is the right time to process it. For a limited amount of operations too young operations can be kept for later and right processing. Last but not least the tuple is written to the KV-store.