BA Step 1 — Architecture overview, Essentials

Tobias Schwan

July 23, 2015

The theoretical foundation is provided by the work of Jacobsen and Adler about defining, creating as well as maintaining materialized views. Furthermore Toshniwal et al. is essential due their work about the stream processing engine Storm.

1 View Maintenance in Web Data Platforms

Jacobsen describes in his article the benefits of view maintenance in distributed storage environments, also called Web Data platforms (WDP). According to him work WDPs are "large-scale data management systems comprising thousands of machines". The quote shows perfectly two unique characteristics of such a system. Unique is on one hand the sheer amount of data successfully managed by the system reflecting the volume aspect in the context of Big Data. On the other hand the aspect of highly distributed environments is unique. They offer high performance as well as scalability through horizontal partitioning of data across storage units.

For the presented work it is important to understand the limitations or better tradeoffs of classical WDPs in order to understand the contribution made here. Jacobsen defines *parallel processing of base and view updates, the lack of ACID transaction support*, and *deferred view maintenance* as core properties of WDPs. Out of these he derives three fundamental challenges regarding consistent view maintenance:

- Concurrent update propagation
- Non-idempotent view update
- Out-of-Order update propagation

2 Dynamic Scalable View Maintenance in Key-Value Stores

On the basis of Jacobsen's work, Adler introduces a View Maintenance System (VMS) based on deferred and incremental view maintenance. His research extend the limited query language capabilities of traditional KV-Stores as well

as maintains up-to-date computed results. Moreover the author introduces a generalized KV-Store model for view maintenance. Based on that model the VMS is introduced which enables efficient and parallel view maintenance in the context of a highly distributed environment. The VMS is designed as an extension to an existing KV-Store. It takes advantage of its core features in order to provide the afore-mentioned additional functionality.

Generally we distinguish *base tables* and *view tables*. Base tables represent the original data whereas view tables are the maintained materialized views.

2.0.1 Architecture

The View Maintenance System consists of three major parts.

- Node extension
- View Manager
- Coordinator

The first component, node extension, exists for each KV-Store node and collects operations affecting the base table. Those operations are read directly from the Transaction log(TL). The TL is a persistent buffer for operations run on the base table. Furthermore it can be seen as a fail-safe mechanism. It is not required for the extension to run on the same machine as the node. When changes to the base table occur, the extension recognizes those from the changes on the transaction log and distributes them to the second component, the view managers. Whereas Node Extensions hold several View Managers to distribute operations to, View Managers can only be assigned to a single extension. This design decision was taken in order to simplify error recovery. View managers are considered a lightweight working unit that is deployed in high numbers. It is the unit of scalability for the VMS since view table updates are applied here. The *1 to n*relation between extension and view manager is managed on a hash-ring. Every extension holds its own ring. Based on the position it determines the responsible view manager and transmits the data.

Mechanism for configuring the VMS are provided by the coordinator, the third and last component.

2.0.2 View Types

The current system can maintain seven view types:

- SELECT
- DELTA
- PROJECTION
- INDEX
- PREAGGREGATION
- Aggregation
  - COUNT
  - SUM
  - MIN
  - MAX
- Joins
  - INNER
  - LEFT
  - RIGHT
  - FULL

The implementation assesses SELECT Statements as follows. Based on the base table key single records uniquely map to one view table records. Dependent

on the evaluation statement a base table is either applied to the view table or not.

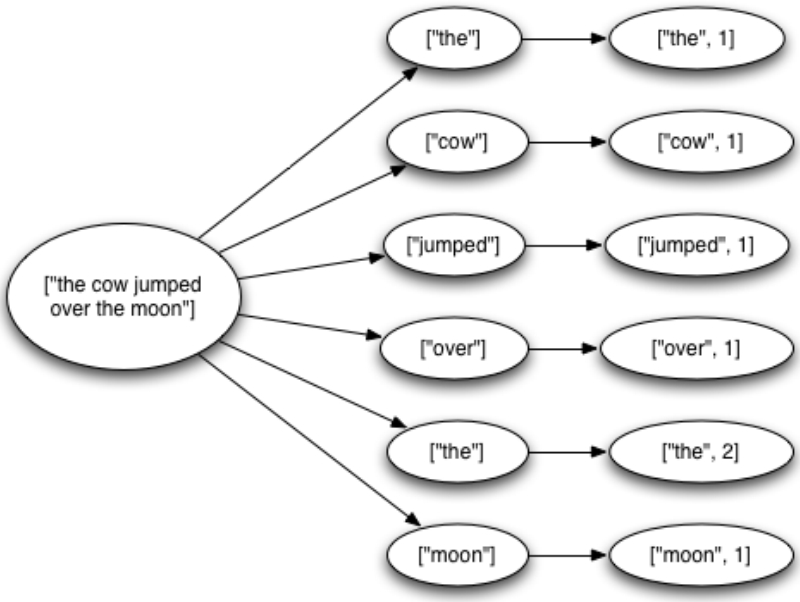
3 Storm @Twitter

Storm was designed as a real-time distributed stream data processing engine with the following requirements:

- Scalable
- Efficient
- Resilient
- Extensible
- Easy to Administer

3.1 Storm

By the authors Storm is seen as a comprehensive combination of existing solutions. In more general terms Storm processes data through a directed graph where vertices represent computation and edges represent the data flow. The authors introduce the terms of *spouts* and *bolds*. Both are seen as vertices in the graph representing computation. The difference between those two lays in their handling of tuples. *Spouts pull* data from external sources whereas *Bolds process* data passed on to them by either Spouts or other Bolds. A complex computation is realized through a chain of Bolds. This chaining of Spouts and Bolds can be illustrated from the perspective of tuples in a tuple tree. The following image shows such a tree for a word-count topology.



3.1.1 Storm's components

A closer look at storms architecture reveals more elements. The so called *Nimbus* handles job management. This includes the distribution of code, assigning tasks to machines as well as monitoring of failures. Therefore it is the gateway between user and cluster. It exists only once.

Topologies represent queries in the context of storm and can be considered as logical query plans from a database systems perspective. Additionally they enable programmers to specify how many instances of verticals inside the graph should exist. All topologies are *Apache Thrift objects*, which provides Storm with compatibility to a variety of programming languages.

Thrift is a project framework to create services that have consistent interfaces and run seamlessly between software parts written in different languages.

Among the cluster topologies are distributed with *Apache ZooKeeper*. Zookeeper is a centralized service for highly reliable distributed coordination. It provides a well-tested implementation for the features of maintaining configuration information, naming, providing distributed synchronization and providing group services.

Additionally to those two components the system contains a third. The *Supervisor* runs on every storm node. All coordination between Nimbus and Supervisor is done using ZooKeeper. After receiving assignments from the Nimbus Worker Processes are spawned by the Supervisor. Worker Processes themselves contain executor threads which run several tasks. A task is an instance of a

spout or a bolt. The Supervisor process monitors health of the worker including respawns if necessary.

3.1.2 Stream Grouping

When processed, a Stream will be partitioned into a number of partitions and split among the bolts' tasks. Therefore only a subset of the tuples from a certain stream is processed by each task. Storm provides several ways to influence this routing behavior, also called grouping:

- Shuffle grouping
- Field grouping
- All grouping
- Global grouping
- Direct grouping
- Local/shuffle grouping
- Custom grouping

Shuffle grouping - Tuples will be spread equally among tasks.

Field grouping - Partitioning is provided based on a field of the tuple. Thus specific tuples can be processed by a single task.

$$\text{hash(fields)} \ \% \ (\#\text{tasks})$$

Where *hash(fields)* is a hashing function. It is not guaranteed that each task is involved in the processing process. As described later, Tasks are the most granular execution unit and represent the actual processing in the Bold.

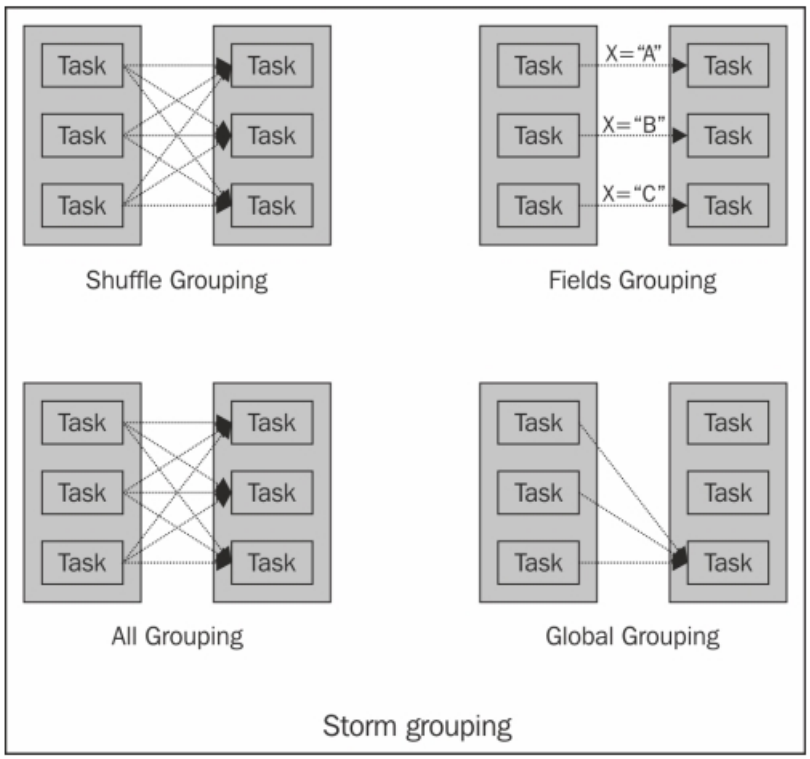
All grouping - replication of tuples rather than partitioning them. Each tuple will be sent to each of the bolt's tasks for processing

Global grouping - sends entire stream to single bolt's task. Usually the one with the smallest ID. Can be used when Bolt is supposed to combine results from previous steps.

Direct grouping - decision where each tuple will be processed is made by the emitter. The emitter declares which tasks of the consumer will receive this tuple. Direct grouping requires a direct stream.

Local/shuffle grouping - minimize network hops in case the target bolt tasks are running on the same worker. Otherwise it will use shuffle grouping.

Custom grouping - own grouping mechanisms can be defined by implementing an internal interface



3.1.3 Lifecycle of a tuple

Out of the box Storm gives a guarantee to process messages. Tuples are considered completely processed once the origin tuple and all derived tuples have been successfully handled. Derived tuples are those who were created by bolds based on the origin tuple. Unsuccessful handling originates from *runtime errors* or *timeouts* - those are failed tuples. They will be replayed by the Storm cluster. In order to track the tuple's state a *unique message ID* is emitted by each spout. This ID is used to send an *ack*nowledgement message once the message is processed completely. In case of additional tuples emitted by bolds, the new tuple will be anchored with the original tuple. Since watching tuple's state as well as the resulting acknowledgement is optional the previously described behavior results in two types of processing semantics. Those are "*at least once*" and "*at most once*". For our given scenario "*exactly-once*" processing semantics are a prerequisite.

3.1.4 Parallelism of a Topology

In order to fine-tune performance of the Storm cluster and make safe assumptions how operations are being processed, it is necessary to understand the ongoing low-level processes. Usually the processing speed of a topology is the

main goal when running a Strom cluster. While this remains true for our scenario, maintaining consistency is our superior goal. Since processing speed is mainly increased through parallelism the order of processing events is essential. Overall there are three setting screws:

- Worker processes
- Executors (threads)
- Tasks

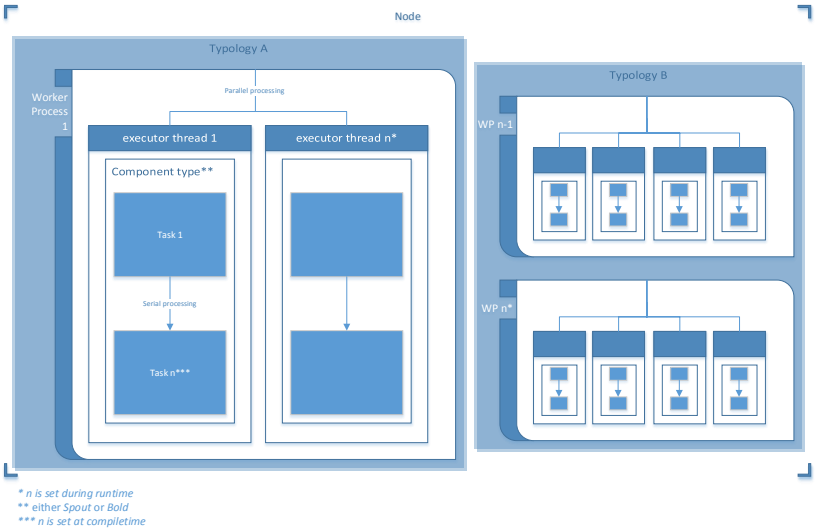
Worker Processes - For each topology a number of Worker Processes exists. These processes are JVM instances running on each node. Their number can by change during the rebalancing phase.

Executors - Executors are spawned by Worker Processes and can be seen as threads in the conventional sense.

Tasks - Tasks are the most granular working unit in Storm. Implemented Spouts or Bolts execute as many tasks across the cluster.

$$\#\text{threads} \leq \#\text{tasks}$$

In the default setting the number of threads equals the number of tasks.



3.1.5 Fault Tolerance

Fault Tolerance is assured through the constant monitoring of both the Supervisor and Nimbus. In case a Worker Process fails it is restarted by the Supervisor.

This restarting behavior is overlooked by the Nimbus. If it fails to start or no heartbeat can be detected, Nimubs will reassign the entire worker to another machine. In case an entire Node fails tasks assigned to that Node will time-out and also reassigned to another machine.

Generally Nimbus and Supervisor daemons are designed to be fail-fast. The author understands under this term the immediate self-destruction of the process whenever unexpected situations are encountered. Furthermore both daemons are kept stateless, since all state related information is either kept on disk or in Zookeeper. Watched by the typical operation system daemon supervisor tools they get restarted without trouble.

It is noteworthy that the nimbus can be seen as a single point of failure in the overall architectural design of Storm. In case the connection to the Nimbus Node is interrupted both exiting workers and supervisors will continue their normal function. Though workers are not reassigned.

3.2 Trident API

The so called Trident API is a higher-level abstraction for doing real time computing on top of storm. It furthermore guarantees exactly-one semantics. The main difference between plain Storm and Trident on top of Storm is the kind of processing they provide. Whereas Storm is a stateless processing framework, Trident extends Storm's functionality to support stateful stream processing. It does that by processing the input as small batches. Therefore the bolt in the Trident topology is replaced with higher-level semantics of functions, aggregates, filters, and states.

Trident functions - modify original tuples with logic. Output will be appended to input tuple. Emitted output tuples that match the input tuple will be removed.

Trident filter - Returns *true* or *false* based on programmable conditions. *False* removes tuple from the stream, otherwise they are passed on.

Trident projections - extracts certain field from tuples and discards the remaining.

Trident aggregate - Aggregation per batch. First, tuples are repartitioned using *global operation* in order to combine all partitions of the same batch into a single partition. All together there are three types available. The *ReducerAggregator*, the *Aggregator*, the *CombinerAggregator*.

Trident partition aggregate - completely replace input tuple with single field tuple. Aggregations work on each partition instead of entire batch.

Trident persistent aggregate -

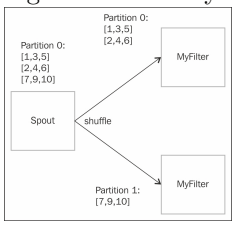
3.2.1 Trident repartitioning

Trident extents the grouping functionality of storm. With Trident there are the following repartitioning operations:

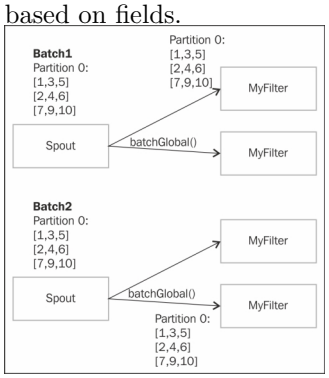
- Shuffle operation
- partitionBy operation
- Global operation
- broadcast operation
- batchGlobal operation
- Custom/partition operation

Any repartitioning operation does not change the content of the tuples. Repartitioning operations are the only situation in which tuples are passed over the network.

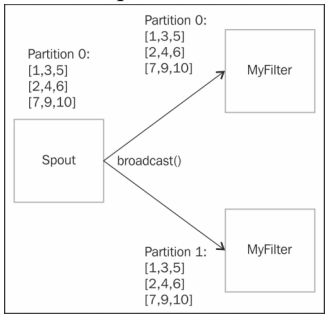
Shuffle operation - distribute processing load uniformly across tasks.



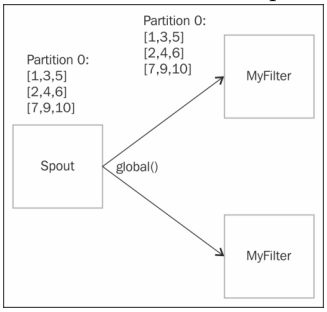
partitionBy operation - partition tuples based on fields.



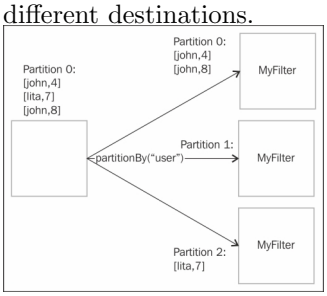
Global operation - route all tuples to the same partition.



broadcast operation - replicate tuples to all nodes instead of partitioning



batchGlobal operation - route all batch tuples to one target task. Other batches of the same stream can have different destinations.



3.2.2 exactly-once semantics

In order to achieve the "*exactly-once*" processing semantics, Trident uses four basic concepts. First, tuples are processed in small batches. Second, each batch has a unique, incrementing transaction ID (taID). In case of reprocessing this ID is reapplied. Third, an order exists within batches. Forth, the Trident spout. It is responsible for creating, maintaining and replaying batches. Based on these concepts we can distinguish the *transactional* from the *opaque transactional* topology. Since using these features is optional one can also implement a *non-transactional* topology. Now, to achieve processing guarantees it is necessary to store both the transactional ID as well as the current value of the field inside of our KV-Store. A simple comparison of the current taID with the one stored in the database gives conclusive insight whether or not to perform the operation. This concept is considered the *transactional* topology in Trident. In order to achieve higher fault tolerance it is necessary to maintain the old value of the field additionally. Then we can consider out topology *opaque transactional*.

3.2.3 The article that claims there is no exactly-once processing

The author claims that in the context of a distributed system it is impossible to have exactly-once message delivery. He sees every setup distributed in which more than one node interact with each other. Considering the problem of the so called *distributed consensus* (Impossibility of Distriuted Consensus with One Faulty Process) he sees the only solution in using only idempotent operation. (Article)

3.2.4 State APIs

Trident ships with all the previously described fault-tolerance logic within the *State API*. The additionally needed fields will be created on the fly by the system without interaction of the developer or user. Furthermore two operations happen automatically in order to improve performance.

1. Read or write operations related to state automatically batch so only the minimum amount of operation will be executed on the database
2. Whenever possible Trident does partial aggregations before sending tuples over the network.

3.2.5 Distributed Remote Procedure Calls

Distributed RPC are used to query on and retrieve results from a Trident topology on the fly. It is managed by the built in distributed RPC server within Storm and acts as middle man between client and topology.

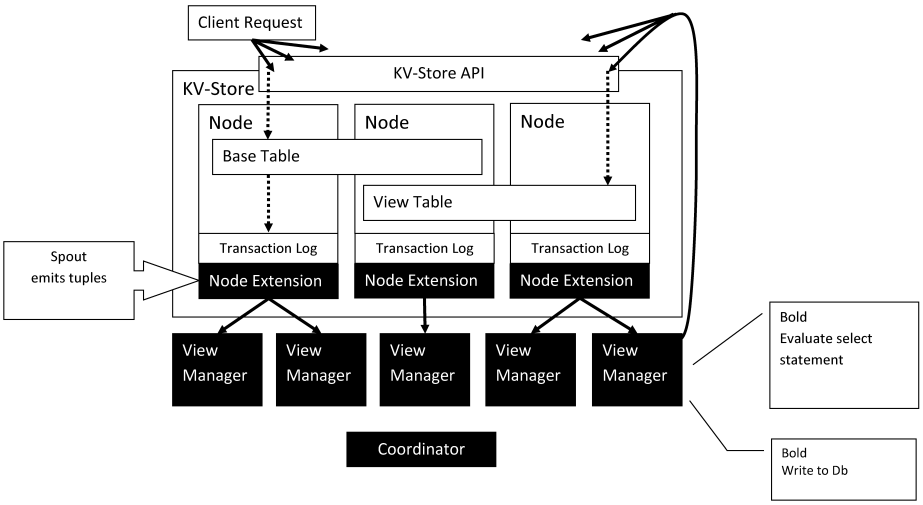
4 Dynamic Scalable View Maintenance in Key-Value Stores through Stream Processing

The main idea of this work is to extend the work of Jacobsen as well as Adler by replacing the existing VMS solution with one based on the principles of stream processing.

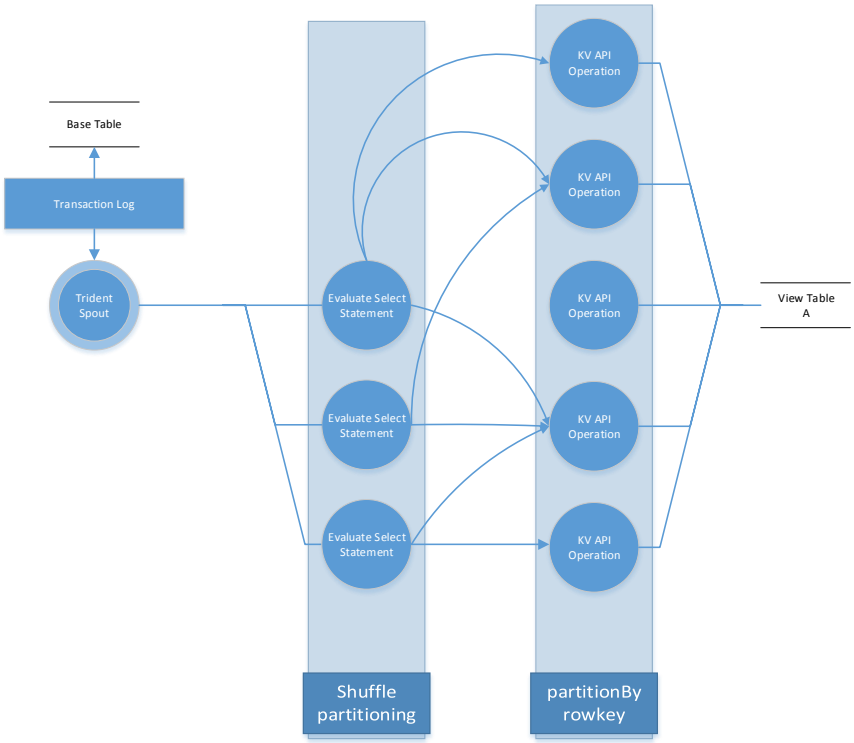
With that in mind the architecture changes as follows:

- The Node extension will be replaced by one Spout per node
- View managers will be replaced by a set of bolds

Whereas swapping node extension and Spout should be less complex as they basically serve the same functionality - emitting data - replacing view managers with bolds will be much more complex. Depend on the view type a whole bunch of bolds will be needed. For the most basic operation SELECT, two bolds will be necessary. The first one evaluates the select statements whereas the second one writes possible data to the database. Since bolds are the unit of scalability those two can be deployed in high numbers and finish the topology fast as well as efficient. For more complex view type more bold-steps will be necessary.



In terms of consistency we can make use of the previously described serial processing of tasks. With using Trident we furthermore guarantee the order of processing batch by batch.


Base Table
Transaction Log
Trident Spout
Evaluate Select Statement
Evaluate Select Statement
Evaluate Select Statement
KV API Operation
KV API Operation
KV API Operation
KV API Operation
KV API Operation
View Table A
Shuffle partitioning
partitionBy rowkey