OVSDB Query Optimizer and Key-Value Interface

OVSCon December 8-10, 2020 | Dmitry Yusupov

NVIDIA

The need for scalability

Elasticity of SDN control plane?

- Kubernetes as of v1.19 supports 5000 nodes in production
- Large topologies with 1000+ EPs, LBs, Namespaces, Policies?
- What about scaling beyond 5000, can we get to 10,000 HVs?

- Can we elastically distribute SDN topology compute and decentralized storage access as cluster growing?

OVN resourcing

Using different SKUs for central and HVs?

- Central components such as NorthD, RAFT OVSDB, CNI master can run on higher performance SKUs

- HV controllers can run on low profile SKUs, e.g. ARM devices with limited CPU and memory

---

- Can be beneficial for large DPU deployments, high-latency Edge IoT networks

A deeper look at OVSDB

Current OVSDB design thoughts

- Emphasis on read I/O scalability with dynamic distributed caches, side effect – stale reads
- Simplistic RAFT-based cluster for HA, side effect – no read after write guarantee, slow writes
- In-memory, unique relational database with only UUID-based query optimizer

Enhanced OVSDB Query Optimizer

Evolution of UUID-based optimizer

- Introduced Primary and Alternate key indexes [1]
- Reusing existing HMAP data structures
- Low overhead - 16 bytes per indexed key
- Results optionally can be ordered

OVSDB Primary key design

Evolution of UUID-based optimizer

- There is no OVSDB schema change
- Using existing per-table “indexes” keyword works well as it has to be unique

```
"Address_Set": {
    "columns": {
        "name": {"type": "string"},
        "addresses": {"type": {"key": "string",
                               "min": 0,
                               "max": "unlimited"}},
        "external_ids": {
            "type": {"key": "string", "value": "string",
                     "min": 0, "max": "unlimited"}}},
    "indexes": [["name"]],
    "isRoot": true},
```

OVSDB Alternate key design

Evolution of UUID-based optimizer

- There is no such construct in OVSDB as of yet
- New boolean flag “alternate_key”: [true|false] introduced
- Alternate key implementation can use b-tree to enable ordered results, e.g. “ordered”:[“asc”|”desc”]

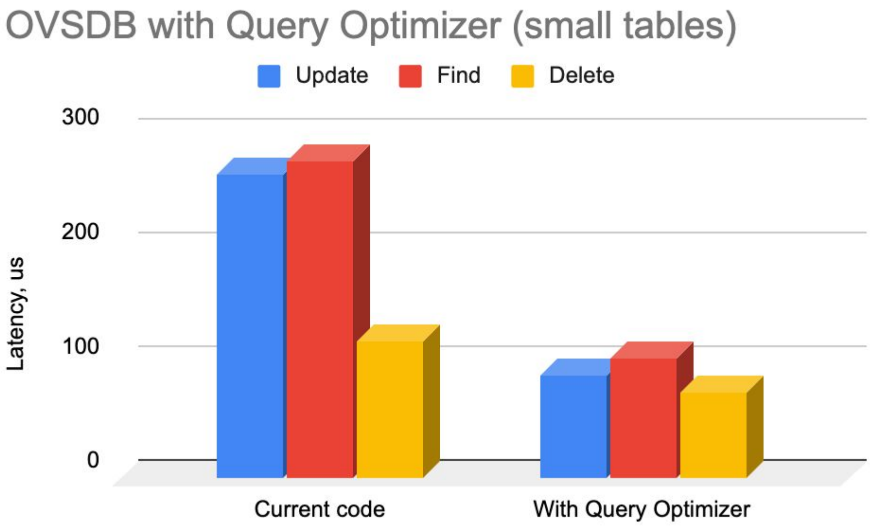
```
"HA_Chassis": {
    "columns": {
        "chassis_name": {"type": "string", "alternate_key":true, "order":"asc"},
```

OVSDB with Primary key performance

Measuring impact of using Primary key with small tables

Table size: 4,000 rows. In microseconds. No RAFT.			
	Update with --may-exist	Find	Delete
Current code	265	277	119
With Query Optimizer	89	104	75

- Linear scan avoided
- O(1) instead of O(N)

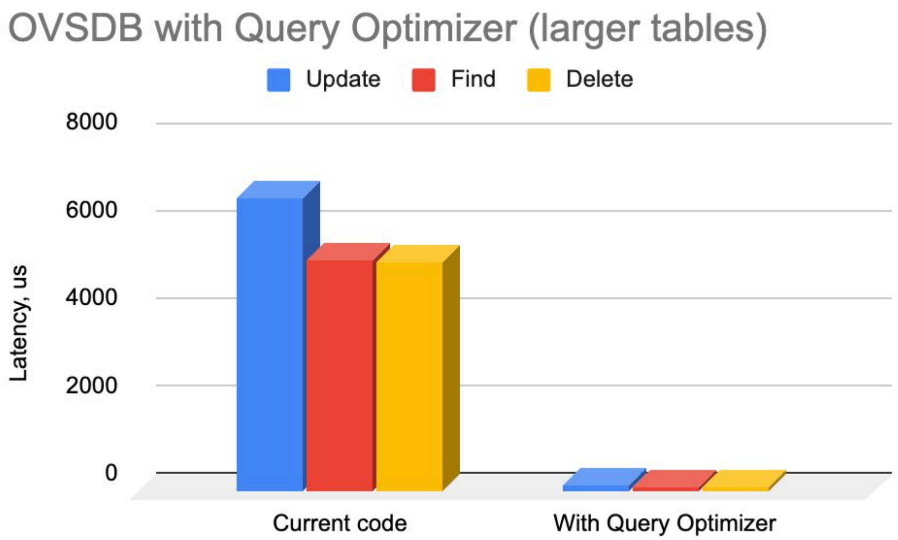


OVSDB with Primary key performance

Measuring impact of using Primary key with larger tables

Table size: 60,000 rows. In microseconds. No RAFT.			
	Update with --may-exist	Find	Delete
Current code	6700	5270	5200
With Query Optimizer	123	105	84

- Linear scan avoided
- O(1) instead of O(N)
- Larger tables bigger impact



Benefits of enabling Query Optimizer in OVN

Primary Key

- Helps with Updates and Selects performance
- OVN Northbound database performance benefits the most
- OVN Southbound database performance improved when custom monitors are used

Alternate Key

- Helps with complex Select queries performance
- Only when user or application executes non-UUID based complex queries

Benefits of enabling Query Optimizer in OVN

If nothing else changed, just Primary key

- Linear scans are O(N) expensive, can we optimize it out a bit? Yes!

- OVN Northbound database can benefit transparently when enabled as below (query_primary would be linear search coverage counter):

```
query_linear              0.0/sec     0.017/sec        0.0028/sec   total: 10
query_uuid               30.8/sec    23.333/sec        0.4628/sec   total: 1666
query_primary            10.4/sec     5.717/sec        0.0972/sec   total: 350
```

Applicability of Query Optimizer

An example of using it with Key-Value interface

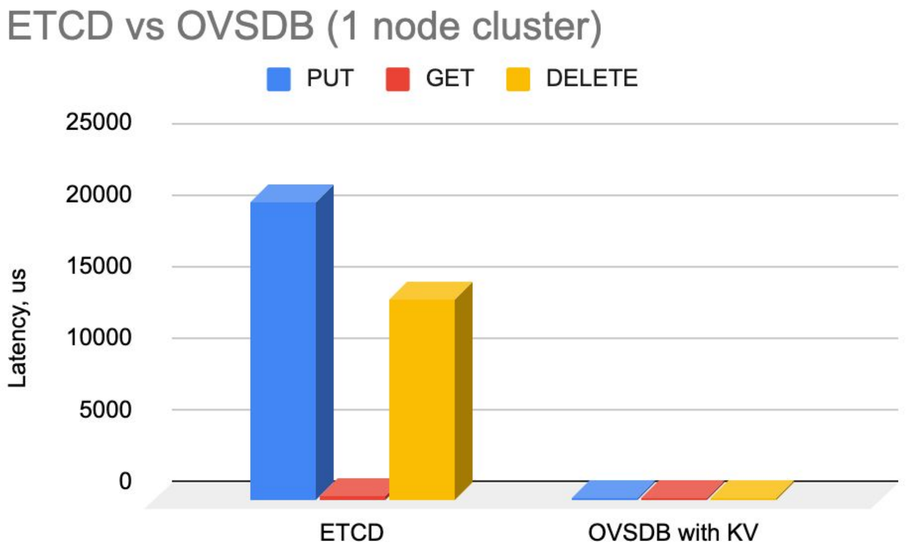
- Benefits of KV interface are in simplicity and scalability
- OVSDB is a great piece of software, so, why not to try?
- OVSKV - a library that is layered on top of libovsdb [2]
- Compatible with ETCD like hierarchical key queries, e.g. /a/b/c* => value

Comparison of ETCD and OVSKV

Using fperf open source performance toolkit [4]

Table size: 60,000 keys. In microseconds.			
	PUT	GET	DELETE
ETCD	20700	230	14000
OVSDB with KV	123	105	84

- 1-node OVSDB
- Using fperf OVSKV backend [3]

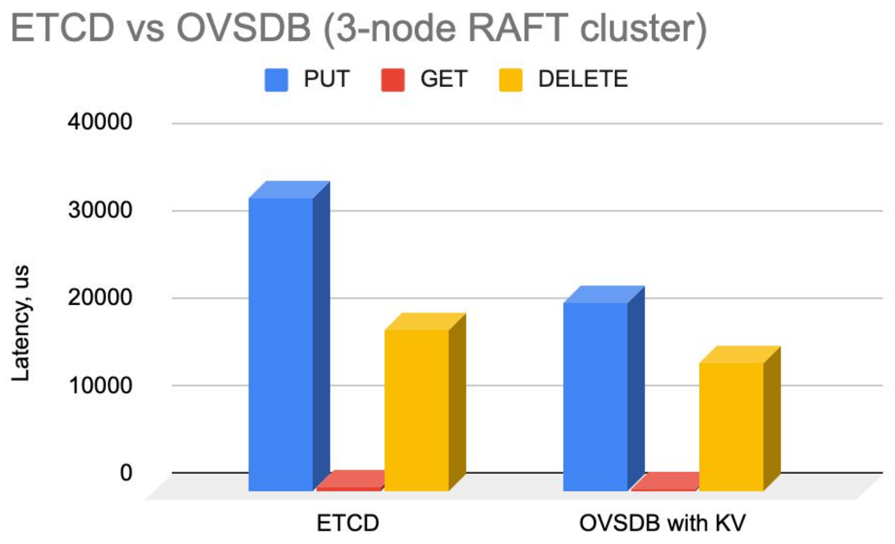


Comparison of ETCD and OVSKV

Using fperf open source performance toolkit [4]

Table size: 60,000 keys. In microseconds.			
	PUT	GET	DELETE
ETCD	33400	415	18300
OVSDB with KV	21400	105	14500

- 3-node OVSDB
- Using fperf OVSKV backend [3]



Future work and ideas

What's next?

- Ordered Alternate key work needs to introduce b-tree implementation
- In the perspective of recent DDlog work, can we scale out computation with D3log?
- Perhaps we can think of introducing multi-writer design to OVSDB?
- What if we switch to Key-Value interfaces? Maybe just for some tables?

Links and References

Show us the code...

1. Primary key implementation OVS github repo
   https://github.com/dyusupov/ovs/tree/query-optimizer-v1
2. OVSKV library github repo
   https://github.com/dyusupov/ovskv
3. fperf for OVSKV backend github repo
   https://github.com/dyusupov/fperf/tree/ovskv
4. fperf github repo
   https://github.com/fperf/fperf