Contribution to Yesquel storage for Web applications

Rui Pacheco[1] and Sergio Escudeiro[2,3]

Instituto Superior Tecnico, Lisboa, Portugal

Abstract. The aim of this document is to study Yesquel database's scalability thoroughly. This is a system that provides performance and scalability comparable to NoSQL with all the features of a SQL relational system. Yesquel's storage uses a new distributed data structure, called YDBT, promising good performance and scaling under contention by many concurrent clients.
We aim to evaluate Yesquel on its architectural choices and benchmark its performance at scale on different workloads capable of exploring its distinguishable features. Finally, we infer conclusions over whether this database system is a viable option as back-end storage for large scale web applications.

Keywords: Yesquel · Universal Scalability Law· Scalable Web Application Storage

1 Introduction

Web applications such as email, online social networks, etc. - need a low-latency back-end storage system to store data. Traditionally, the storage system chosen had been a SQL database system, which is convenient for the developer but imposes a scalability bottleneck. Two movements arose, to shed functionality by using a scalable NoSQL system or to use *application-level techniques* (e.g., manual caching) to improve scalability in an SQL database system. However, shifting to a NoSQL system involves changing the data model and application-specific queries to the new specific NoSQL system API (which defers from the previously implemented SQL relational system).

As a solution to this challenge, Yesquel presents itself as a storage system that provides all of the features of a SQL relational database (zero to none changes to the application logic are needed) , but scales as well as NoSQL on NoSQL workloads.

The git commit identifier that should be considered for evaluation purposes is: "finalv2" and the correspondent release is: "first delivery 2".

2 Architectural Overview

2.1 SQL System

A SQL database has two main components: a query processor and a storage engine. A user, or an applications program, interacts with the query processor and the query processor, in turn interacts with the storage engine. Essentially, the query processor receives an instruction written in Structured Query Language (SQL), chooses a plan for executing the instructions and carries out the plan by issuing requests to the storage engine responsible for manipulating data. It is worth remarking that both components typically lie on a SQL server cluster. Therefore, to scale SQL, we must scale both components simultaneously.

2.2 Yesquel

Yesquel brings a new approach to the traditional architecture by shifting the query processor workload from a server to the client. Resulting in each client having its own query processor so that, as the number of SQL clients increases, the SQL processing capacity also increases.

Scaling the storage engine is a challenge as it must serve requests by many query processors, possibly operating on the same data - contention. To tackle it, Yesquel's storage engine uses a turbined version of a *distributed balanced tree* (DBT). The tweaked DBT in Yesquel, called YDBT, is maintained by a cluster of servers (each storing a subset of ramifications) and has features that are targeted for Web workloads.

First, YDBT balances load on the servers even if the workload is skewed (non-uniformly distributed) and even if it changes over time, it does so, by dynamically partitioning and redistributing the tree according to specific node workloads.

Second, data can be efficiently inserted, searched, deleted, and enumerated in order, often in only one network round trip. It does so by utilising a new caching scheme and new mechanisms and policies to split nodes.

Third, the data structure provides strong consistency and fault tolerance supporting ACID transactions (fulcral to SQL systems) by relying on new ways to handle and avoid contention, and a new approach to tree snapshots.

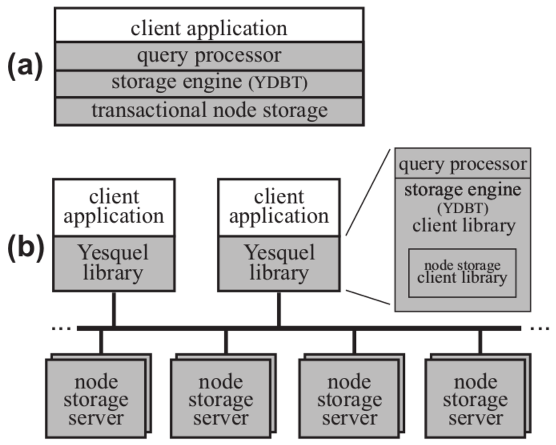

Fig. 1. Logical (a) and physical (b) architecture of Yesquel, with its components shown in gray.

3 Scalability

3.1 YDBT - Yesquel's trick up it's sleeve

To scale, Yesquels storage engine needs to partition the data of an ordered map across the storage servers. Doing so requires solving the following five challenges:

- Provide locality, so that neighbouring keys are often in the same server, allowing them to be fetched together during enumeration.
- Minimize the number of network round trips for locating keys.
- Balance load by spreading keys across servers, even if the workload is non-uniform and changes over time. This requires continuous and adaptive re-balancing without pausing or disrupting other operations.
- Provide reasonable semantics under concurrent operations. For example, while a client changes an ordered map, another client may be enumerating its keys, and the result should be sensible.
- Provide good performance under concurrent operations.

To solve them, yesquel relies on the following techniques:
Locality is made possible by organising the data as a tree where consecutive keys are stored in the leaves
To **Minimise Round Trips**, it uses an optimistic caching scheme with a new technique called back-down search. With this combination, clients search the tree in two steps: the first goes upwards toward the root, the second downward toward the leaves.
To **Balance Load**, YDBT splits its nodes as in balanced trees, but with different policies and mechanisms to split. The policy is based on load, and the mechanism is based on transactions and a technique called *replit* that is an hybrid of splitting and replication.
To **Provide Reasonable Semantics** under concurrent operations, YDBT provides snapshots. Thus, a client can enumerate keys using an immutable snapshot, while other clients change the map.
To **Provide Adequate Performance** under concurrent operations, YDBT introduces techniques to avoid conflicting updates to the tree.

3.2 Decomposing a request into a pipeline of stages

Each client application has its own query processor, to which it submits SQL queries. The query processor creates a plan of execution transforming the queries into operations on tables and indexes in the storage engine. Afterwards, the Yesquel storage engine carries out the plan by traversing the **Yesquel *distributed balanced tree*** to read/insert/delete affected nodes. It is important to note that all the components except the YDBT are present on the client host itself.

3.3 Universal Scalability Law (USL)

In light of the USL, we'll explore the 3 different factors contributing to the system's scalability.
Performance coefficient is the computational work that is divisible between the available components. In yesquel, there are three components that contribute towards this division:
The query processor and the Yesquel storage engine are both on the client host, therefore, as more clients join in, the system's theoretical query processing capacity also increases as both the formulation of the execution plan and its execution is the client's computational responsibility.
The *distributed balanced tree* responsible for storing the database, scattering it across the storage servers. Furthermore, as the scattering is dynamic and reshapes to the specific load it is receiving, the computational burden from the queries is concurrently distributed.
Contention is the portion of the work can only be executed serially. In the context of databases, within the ACID properties of transactions, there is a property called Atomicity. It states that a transaction must fully complete, saved (committed) or completely undone (rolled back). In workloads where multiple concurrent transactions touch the same relations, a high incidence of roll backs is expected, resulting in a serial portion of work happening. Within the yesquel's architecture, we've identified this as a possible contention contributor. Fortunately, as the designers of this brilliant database demonstrate, such does not occur **oftenly** thanks to three techniques: Multi-version concurrency control, which aborts transactions at any signs of contention; Delegated splits where clients delegate the coordination of the tree split to remote splitter processes, each is responsible for serialising the splits to a disjoint set of nodes not affecting ongoing transactions; Mostly commutative operations so that two parties can often modify the same node concurrently without conflicts (hence without aborting their transactions).
That said, it does occur in specific workloads (different from yesquel's workload scope - web application database) that operate on lots of data on data like analytics queries, resulting in compulsive *distributed balanced tree* cache invalidation in each of the clients as the tree structure fluctuates a lot. The compulsive invalidation and re-fetching of the tree cache on each client is a serial operation on the storage server's cluster as it requires an holistic snapshot.
Crosstalk, also called the consistency or coherency penalty, potentially happens between each pair of components in a system. In yesquel, there is crosstalk between the storage servers whenever a distributed balanced tree splits in result of a new insertion/modification, deletion and when duplication of a tree subset occurs. Crosstalk also occurs when duplicated tree subsets are modified, incurring in update messages between the storage servers.

4 Evaluation

4.1 Test bed

Our test bed is a cluster of Docker Containers each running an Yesquel storage server and communicating in a local shared virtual network. A second cluster of clients issuing SQL requests is running on the host machine, therefore, this cluster will be on a different virtual network. All containers are hosted on a single host running Ubuntu 16.04. Each container is running Ubuntu 16.04 (v4.13.0-45-generic kernel). The ping round-trip latency between the containers and between clients and containers is close to 0 ms as they are in the same host.

4.2 Workloads

We've devised a very simple workload of simultaneous reads from a database with 1 table with 2 integer columns populated with 2 relations. Six different scenarios were devised:

1. 1 client issuing requests to a single yesquel storage server
2. 10 concurrent clients issuing requests to a single yesquel storage server
3. 30 concurrent clients issuing requests to a single yesquel storage server
4. 1 client issuing requests to two yesquel storage servers working together
5. 10 concurrent clients issuing requests to two yesquel storage servers working together
6. 30 concurrent clients issuing requests to two yesquel storage servers working together

We've recorded the response time for 0,10000,100000,1000000 and 10000000 requests per client.

4.3 Results

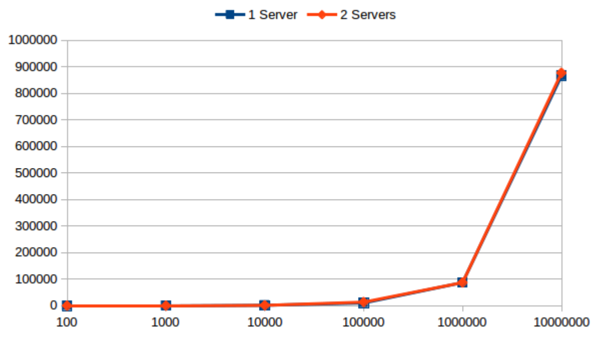

Fig. 2. Results for workloads 1 and 4, where X is time in microseconds.

5 Conclusion

Unfortunately, all results show no improvement from adding a second yesquel storage server to the cluster (all the other workloads are exactly the same - no improvement). We suspect this might have been caused by our very simple database population method of only two relations. Nevertheless, given how the dynamic distributed tree load balancing within yesquel works, we expected that the accessed branches were duplicated in the second server, effectively scaling with the increased workload. That said, we can also justify such results because the time it took to read a relation from yesquel storage servers involved retrieving 2 integer, meaning it was instantaneous, so the difference in replicating a sub-tree might not be observable through our chosen workload. In the future, we'll explore different, more suited workloads hopping to substantiate our claims regarding the Universal Scalability Law.

References