# Yesquel: scalable SQL storage for Web applications

Marcos K. Aguilera* Joshua B. Leners† Ramakrishna Kotla* Michael Walfish‡
*unaffiliated †UT Austin ‡NYU

8 November 2014

arXiv:1411.2160v1 [cs.DC] 8 Nov 2014

## Abstract

Based on a brief history of the storage systems for Web applications, we motivate the need for a new storage system. We then describe the architecture of such a system, called Yesquel. Yesquel supports the SQL query language and offers performance similar to NOSQL storage systems.

## 1 Introduction

Web applications (web mail, web stores, social networks, etc) keep massive amounts of data (account settings, user preferences, passwords, emails, shopping carts, wall posts, etc). The design of such an application often revolves around the underlying storage system.

This paper briefly describes a new storage system for Web applications, called Yesquel. Yesquel combines several advantages of prior systems: it supports the SQL query language to facilitate the design of applications, and it offers performance similar to NOSQL key-value storage systems.

To achieve these goals, Yesquel leverages techniques for scalability and fault tolerance previously used in NOSQL systems, and uses them to obtain a SQL system. In addition, Yesquel incorporates a unique architecture that provides an embedded query processor to each of its clients, that implements distributed transactions at a low level, and that builds a distributed index structure on top of such transactions. This architecture is accompanied by new and efficient mechanisms to execute transactions and to store database tables and indexes.

This paper focuses on Yesquel's motivation and architecture. A subsequent paper will describe Yesquel's transactions and storage engine.

## 2 Historical perspective and motivation

The storage systems used in Web applications have evolved dramatically over the past 25 years, and the history brings interesting insights. We can roughly divide these storage systems into four generations.

- *First generation: file systems.* In the early 1990s, Web pages were static. Web servers received HTTP requests for a file, read the file from their local file system, and returned it to the user. The storage system for Web applications was simply the file system holding such files.

- *Second generation: file and* SQL *database systems.* In the mid to late 1990s, the Web saw the emergence of *dynamic content*—content that depends on who the user is or what the user has done (e.g., the items in a shopping cart). To generate a page, the Web server invoked a program written in languages such as Perl, Python, PHP, Java, etc. The program stored data needed to generate the page in a central SQL database system. The use of SQL was convenient, because SQL has many useful features (joins, secondary indexes, transactions, aggregations, many data types, etc). The storage system, thus, was a combination of the file system for static content and the database system for dynamic content.

- *Third generation: highly scalable systems.* In the 2000s, large Web sites emerged, such as Amazon, Hotmail, Google, Yahoo, and others. These sites had a rapid growth in the number of users; soon the database system became a performance and scalability bottleneck. Scaling the database system was hard or expensive, so developers decided to replace the SQL database system with their own home-grown storage systems, such as the Google File System [9], BigTable [4], Dynamo [7], and others. This was the

beginning of a movement that later became known as NOSQL.

- *Fourth generation: cloud storage systems.* In the 2010s, many Web applications moved to the cloud, where many vendors share the same computing and storage infrastructure. Storage systems were designed not just to scale, but also to provide isolation, so that applications do not interfere with one another. Examples of such storage systems include Amazon S3, SimpleDB, Azure Blobs, and Azure Tables.

A highlight in this history is the emergence of the NOSQL movement, ten years ago, which sought to replace the SQL database system with cheaper custom-built alternatives. These alternatives were much simpler than a SQL database system, and thus were easier to scale, but they offered more restricted functionality: few NOSQL systems offer transactions, secondary indexes, joins, or aggregations, and certainly no NOSQL systems offer all the features found in SQL. Thus, the move from the SQL database system to NOSQL systems came at the cost of functionality.

Today, there are several dozens of NOSQL systems, each offering a different set of features, and each with its own application interface. Web application developers face a difficult choice of what storage system to use, because it is unclear a priori what features the application might need. Even if developers can identify an appropriate system, the application may evolve and later require functionality from the storage system that was not originally deemed useful. Because each storage system provides its own interface, once an application is developed to use one storage system, it becomes hard to replace it with another storage system should the need arise—a problem known as vendor lock-in.

We start with the observation that NOSQL is not a feature but rather the absence of a feature. What is the NOSQL movement really trying to achieve? The answer is a distributed storage system that is scalable; fault tolerant; simple and sane to design, implement, and maintain; and nimble and cheap to run.

We took these characteristics as our goal in developing Yesquel, but we also wanted to provide full support for the SQL query language. Besides having a rich set of features, SQL is a well-known query language (it is taught in database courses), and it is an industry standard, so applications developed for SQL avoid the problem of vendor lock-in.

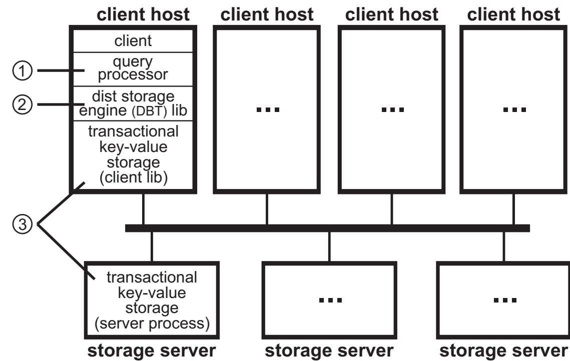


Figure 1: Architecture of Yesquel.

# 3 Architecture

A SQL database system has two main components: a query processor and a storage engine. The query processor parses and executes SQL queries, while the storage engine stores tables and indexes.

The architecture of Yesquel is depicted in Figure 1. Each client has its own embedded query processor (box ① in the figure), which is a user library that links against the client application. As a result, as the number of clients increases, the number of query processors also increases proportionately. This property allows Yesquel to scale the query processing capacity of the system. The query processors all share a common storage engine.

For this idea to work, the storage engine must be designed to handle a large number of query processors, which at a given time may try to access the same tables and indexes. Handling concurrent access to such data is a key challenge addressed by Yesquel.

In Yesquel, the storage engine is implemented as a *distributed balanced tree (DBT)* (box ②). A DBT is a balanced tree data structure whose nodes are distributed over a set of storage servers. The reason for distribution is to scale the performance of the DBT; in Yesquel, this is done by increasing the number of storage servers.

The Yesquel DBT library is implemented on top of a transactional key-value storage system—a simple transactional system that stores key-value pairs on the storage servers (boxes ③). That is, the nodes of the Yesquel DBT are stored as key-value pairs in the key-value storage system. This design separates the implementation of the DBT from the implementation of the distributed transactions. The transactions in the key-value storage system provide *snapshot isolation* [3], a property often used in commercial database systems. Because the DBT is implemented above the transactions, the DBT benefits from the full power of transactions; for example, the Yesquel DBT uses

transactions to atomically move data across DBT nodes to balance load.

Note that transactions are provided at the lowest layer—the key-value storage system, which is the layer that actually stores the data bits on storage servers. As a result, the transactional protocol enjoys greater efficiency. Specifically, Yesquel uses multi-version concurrency control—the most sophisticated form of concurrency control—which requires managing multiple versions of each data item, and this can be done most efficiently and effectively at the layer that stores the actual data.

# 4 Related work

F1 [13] is a distributed storage system designed for Google's AdWords; it provides support for SQL with a non-relational hierarchical model. F1 has a different architecture from Yesquel: F1 is layered on top of the Spanner system, which in turn is layered onto a Bigtable-based implementation. Spanner [5] provides distributed transactions, while Bigtable provides the DBT functionality. Thus, the DBT is implemented *below* transactions. In contrast, in Yesquel distributed transactions are provided at the lowest level (the key-value storage system), and the DBT is implemented *above* the transactions. As explained, this choice allows the DBT to leverage transactions. Moreover, Yesquel provides an embedded query processor to clients. Yesquel also uses different protocols for transactions, which unlike F1 do not require special hardware clocks. However, F1 provides a stronger transaction isolation property (strict serializability) than Yesquel, and F1 works in a geo-distributed deployment, whereas Yesquel runs within a single data center.

MOSQL [15, 16] is a distributed storage engine for MYSQL. MOSQL has a different architecture from Yesquel: its transactional layer runs on top of (1) a distributed storage layer without transactions, and (2) a certifier service implemented as a replicated state machine. Atop the transaction layer, MOSQL has a B+tree. In contrast, Yesquel provides distributed transactions at the lowest level, Yesquel uses a more sophisticated DBT, and Yesquel does not use a logically centralized certifier. As a result, we believe Yesquel is more efficient and scalable than MOSQL. However, MOSQL provides a stronger transaction isolation (serializability) than Yesquel.

Traditionally, there are two broad architectures for a distributed SQL database system: shared-nothing and shared-disk [11]. Each has advantages, and there is a long-running debate about the two approaches, with commercial vendors supporting one or another, and sometimes both.

In *shared-nothing* systems (Clustrix, Greenplum, H-Store/VoltDB [10], IBM DB2 DPF, Microsoft SQL Server, MySQL Cluster, Netezza, Tandem NonStop, Teradata, etc.), each database server stores part of the data; the system decomposes queries, executes the sub-queries at the appropriate servers, and combines the results for the client. The benefits of this architecture can be substantial; for example, sub-query processing happens close to the data, thereby avoiding network communication. However, performance crucially depends on a good partition of the data, and such a good partition may not exist (if the query set is dynamic). Or a partition may be expensive to identify [14]: the classical approach is to rely on a (well-paid) database administrator to partition manually. Software can help (as in H-Store/VoltDB [10, 12]) but not in all cases.

In *shared-disk* systems (IBM DB2 pureScale, Oracle RAC, ScaleDB, Sybase ASE Cluster Edition, etc.), every database server can access every data item; the disks are shared over the network. The advantage is easier management as there is no need to carefully partition the database. However, in this architecture, servers must carefully coordinate access to storage; the traditional solutions use distributed locks, distributed leases, and cache coherence protocols, which bring complexity and cost.

A recent movement called NEWSQL advocates new architectures for database systems. NEWSQL systems include H-Store/VoltDB and Hekaton. H-Store/VoltDB is an in-memory shared-nothing system, where each server runs a single thread to avoid synchronization overheads. Similar to other shared-nothing systems, performance critically depends on a good partition of the data. Hekaton is a centralized in-memory database system that features a lock-free index structure; it performs well, but scalability is limited to a single machine.

Finally, there has been much work on Big Data systems that process massive data sets at many hosts (e.g., [1, 2, 6]). Some of these systems support SQL [17, 8]. However, they are intended for data analytics applications and are not directly applicable to Web applications.

# 5 Summary

We described the architecture of Yesquel, a storage system designed for Web applications, whose workload consists of relatively small queries that must execute quickly. Yesquel provides full support for SQL and offers performance similar to NOSQL key-value storage systems. Basically, Yesquel maps SQL queries to operations on a DBT, which in turn are mapped to operations on a transactional key-value storage system. In other words, Yesquel inter-

nally leverages a NOSQL storage system to provide its scalability.

# References


[1] http://hbase.apache.org.

[2] http://hadoop.apache.org.

[3] H. Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, pages 1–10, May 1995.

[4] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218, Nov. 2006.

[5] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation*, pages 251–264, Oct. 2012.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.

[8] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, Aug. 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[10] R. Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, Aug. 2008.

[11] K. D. Levin and H. L. Morgan. Optimizing distributed data bases: a framework for research. In *National computer conference*, pages 473–478, May 1975.

[12] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *International Conference on Management of Data*, pages 61–72, May 2012.

[13] J. Shute et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, Aug. 2013.

[14] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, Mar. 1986.

[15] A. Tomic. *MoSQL, A Relational Database Using NoSQL Technology*. PhD thesis, Faculty of Informatics, University of Lugano, 2011.

[16] A. Tomic, D. Sciascia, and F. Pedone. MoSQL: An elastic storage engine for MySQL. In *Symposium On Applied Computing*, pages 455–462, Mar. 2013.

[17] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *International Conference on Management of Data*, pages 13–24, June 2013.