Yesquel: scalable SQL storage for Web applications

Marcos K. Aguilera
VMware Research Group

Joshua B. Leners
UT Austin, NYU

Michael Walfish
NYU

Abstract

Web applications have been shifting their storage systems from SQL to NOSQL systems. NOSQL systems scale well but drop many convenient SQL features, such as joins, secondary indexes, and/or transactions. We design, develop, and evaluate Yesquel, a system that provides performance and scalability comparable to NOSQL with all the features of a SQL relational system. Yesquel has a new architecture and a new distributed data structure, called YDBT, which Yesquel uses for storage, and which performs well under contention by many concurrent clients. We evaluate Yesquel and find that Yesquel performs almost as well as Redis—a popular NOSQL system—and much better than MYSQL Cluster, while handling SQL queries at scale.

1. Introduction

Web applications—email, online social networks, e-commerce, photo sharing, etc.—need a low-latency backend storage system to store critical user data, such as emails, account settings, public postings, shopping carts, and photos. Traditionally, the storage system had been a SQL database system [48], which is convenient for the developer [34]. However, with the emergence of large Web applications in the past decade, the SQL database became a scalability bottleneck. Developers reacted in two ways. They used application-specific techniques to improve scalability (e.g., caching data in a cache server to avoid the database [46], partitioning the data in the application across many database instances)—which is laborious and complicates application code. Or they replaced database systems with specialized NOSQL systems that scaled well by shedding functionality.

However, NOSQL has three issues. First, because NOSQL systems drop functionality, they shift complexity to the application. Second, there is a plethora of systems (Figure 1), with different subsets of features. A developer can have a hard time deciding which system is best suited for the application at hand: it may be unclear what features are needed now, what might be needed tomorrow, what data model to use, etc. By contrast, SQL has a broad feature set (transactions, secondary indexes, joins, subqueries, etc.). Third, each NOSQL system has its own specialized interface, which creates a lock-in problem: once an application is developed for one NOSQL system, it is hard to replace it with another NOSQL system. By contrast, SQL has a well-known industry-standard interface.

In this paper, we propose Yesquel, a storage system that provides all of the features of a SQL relational database, but scales as well as NOSQL on NOSQL workloads. Yesquel is designed for serving Web applications, not as a general-purpose database system. It introduces an architecture that is well suited for Web workloads. (Section 3 explains why Yesquel does not scale as well under other workloads.)

A SQL database system has two main components: a query processor and a storage engine. The query processor parses and executes SQL queries; it is an active component that drives the computation of the query. The storage engine stores tables and indexes; it is a passive component that responds to requests from the query processor to manipulate data. To scale SQL, we must scale both components. Yesquel does that through a new architecture, which scales the query processor using a well-known technique: parallelization of computation [19, 55, 60, 67, 68]. Each client gets its own query processor (Figure 2) so that, as the number of SQL clients increases, the SQL processing capacity also increases. Scaling the storage engine is harder; this is a key technical challenge addressed by this paper. The problem is *contention*: the storage engine must serve requests by many query processors, possibly operating on the same data, and it must return fast and sensible results. To tackle this challenge, Yesquel introduces a new scalable distributed data structure for sharding data.

Specifically, the Yesquel storage engine uses a new type of *distributed balanced tree* or DBT [2, 42, 61]. A DBT spreads the nodes of a balanced tree across a set of servers, while ensuring that consecutive keys are often stored together at the leaves. The DBT in Yesquel, **YDBT**, has features that are targeted for Yesquel and its Web workloads.

First, YDBT balances load on the servers even if the workload is skewed and even if it changes over time. This is needed to ensure that Yesquel scales well in a variety of settings, not just when access is uniformly distributed. Second, data can be efficiently inserted, searched, deleted, and enumerated in order, often in only one network round trip.

SOSP'15, 4–7 October 2015, Monterey, CA.

http://dx.doi.org/10.1145/2815400.2815413

This is useful to efficiently perform sorting, joins, unions, intersections, range queries, and other common operations of SQL. Third, the data structure provides strong consistency and fault tolerance. This is expected of a database system. Fourth, operations can be grouped and executed as ACID transactions. This is needed to implement SQL transactions.

YDBT introduces a new caching scheme, new mechanisms and policies to split nodes, new ways to handle and avoid contention, and a new approach to tree snapshots (§4).

We evaluate the performance of YDBT (§6). Compared to previous DBTs, YDBT eliminates or drastically reduces scalability bottlenecks, eliminates disruption from snapshots, and improves transaction throughput by up to 25×. We also compare Yesquel to Redis, a popular NOSQL system. On NOSQL workloads, Yesquel's throughput is at most 21% worse than Redis's under reads, updates, or scans; and Yesquel scales well up to 48 servers—the maximum we could experiment with—with a scalability factor (improvement over one server) of at least 40. Finally, we test Yesquel on the SQL queries of Wikipedia and find that Yesquel scales its throughput well up to 48 servers, with a scalability factor of 41.2, a throughput of 95K pages/s (≈8 billion pages a day), and a 99-percentile latency of 69 ms. The cost of these benefits is that (1) Yesquel consumes additional network bandwidth, memory, and client-side CPU, and (2) Yesquel underperforms traditional database systems on certain workloads outside its target domain, such as data analytics with long-running queries that access many rows.

Yesquel's results show that, contrary to various claims made about the inevitability of NOSQL, SQL storage can offer performance and scalability similar to NOSQL for Web applications.

2. Overview

2.1 Setting and goals

Web applications in datacenters have multiple tiers. The Web front-end tier communicates with end users; the middle (or application) tier provides the application logic; and a storage tier holds persistent user data (account settings, wall posts, comments, etc.). This paper's focus is the storage tier. Until the late 1990s, this tier consisted of a SQL database system; as applications grew, this component became a performance bottleneck, and NOSQL emerged as a scalable alternative, but with limited functionality.

Our goal is to design a SQL storage system that scales well when used to serve Web applications. We do not wish to design a general-purpose database system. Indeed, Stonebraker et al. [64] have advocated against such one-size-fits-all solutions, by showing that they can be dramatically outperformed by designs optimized for a domain.

Web applications have lots of data and users. Queries used to serve Web pages are *small and numerous:* each query touches a small amount of data, but because there are many simultaneous queries, in aggregate they access large amounts of data. Queries are *latency-sensitive*, as Web pages must display quickly (less than a second [50]). These characteristics contrast with data analytics applications, where queries are long (a single query may run for hours or days) but there are relatively few of them, and they can tolerate seconds of setup time.

system	data model	durability	distributed transactions	secondary indexes	joins	aggregation
Sinfonia	bytes	yes	lim	no	no	no
COPS/Eiger	kv	yes	lim	no	no	no
Dynamo	kv	yes	no	no	no	no
Hyperdex	kv	yes	yes	yes	no	no
Memcached	kv	no	no	no	no	no
Riak	kv	yes	no	yes	no	no
Scalaris	kv	no	yes	no	no	no
Redis	kv	yes	no	no	no	no
Voldemort	kv	yes	no	no	no	no
Walter	kv	yes	yes	no	no	no
AzureTable	table	yes	no	no	no	no
BigTable	table	yes	no	no	no	no
Cassandra	table	yes	no	lim	no	no
HBase	table	yes	no	no	no	no
Hypertable	table	yes	no	lim	no	no
PNUTS	table	yes	no	no	no	no
SimpleDB	table	yes	no	yes	no	yes
CouchDB	doc	yes	no	yes	no	lim
MongoDB	doc	yes	no	yes	no	lim

Figure 1—A partial list of NOSQL systems. In the data model column, "kv" stands for key-value, while "doc" stands for a document data model. Other columns are typical features of SQL systems; "lim" means the feature is available partially or requires programmer assistance.

We target deployment within a datacenter. We assume that the datacenter has a low-latency, high-bandwidth network, and that computers do not fail in Byzantine ways (but computers may crash). We assume that clocks are often synchronized fairly closely (via NTP, for example), but we do not assume that this is always true; clocks can drift.

2.2 Drawbacks of existing scaling techniques

There are currently two broad approaches to scaling a Web application. One approach is to retain the SQL database and use *application-level techniques* to lessen or spread the load on the database. With *manual caching*, the developer makes the application cache the results of a SQL query (e.g., in memcached [46]) so that subsequently it can lookup the result in the cache if the query recurs. With *manual database sharding*, the developer divides the data set into several disjoint databases (for example, usernames that begin with A–N and usernames that begin with O–Z); the application code is aware of the division and manually selects the appropriate database to execute a query. With *denormalization*, applications break down a database's relational structure, to avoid queries that touch multiple tables. For example, to avoid joins, the developer may store multiple attributes in a single column (e.g., as a comma-delimited string), or she may replicate the columns of one table into another.

Application-level techniques are effective and widely used today. However, they transfer complexity to the application, they are error-prone, and they are limited. With manual caching, the developer must carefully choose what and when to cache; also, when data changes, the application must invalidate the cache manually or live with stale results. And while there are, intriguingly, caches that maintain themselves automatically [31, 52, 57], they require application-specific tuning or do not support all of SQL. With manual database sharding, rebalancing is difficult and there is little support for queries across databases. With denormalization, the developer must manually ensure that the replicas or the result of joins remain correct when data changes.

Application-level techniques are complementary to our approach. By making the SQL database more scalable, we can lessen or avoid their use, simplifying the application. Arguably, this shifts the scalability burden and complexity to Yesquel, but this is sensible because Yesquel can be designed and implemented only once while benefiting all its Web applications.

The second broad approach to scale the storage system is to replace the SQL database with one of a large selection of NOSQL systems (Figure 1). However, NOSQL has significant disadvantages, as noted in the Introduction.

2.3 Architecture of a database system

We give a brief overview of the architecture of a database system; we refer the reader to [29] for a thorough treatment. A SQL database system has two core components:

- The *query processor* takes a SQL query, parses and compiles it into a query plan, and executes the plan by issuing requests to the storage engine.
- The *storage engine* executes requests to read and write rows of tables and indexes, to traverse them, and to start and commit transactions.

Database systems provide a modular interface between these two components. In MYSQL, for example, this interface is exposed to developers, which has led to many storage engines (MyISAM, InnoDB, ndb, etc.) that work with the same query processor. At the core of the storage engine is a data structure used to store tables and indexes. In a central database system, this structure is typically a B-tree.

2.4 Key idea and challenges

Yesquel scales the database system through its architecture. The basic idea is to supply a query processor per client (a client is a Web application process) and design a storage engine that can handle all query processors. For this to work, the storage engine must be scalable and highly concurrent.

Different storage engines have different capabilities, such as transactions and indexing, which affect the features and efficiency of the database system. To support all SQL features efficiently, a common choice is for the storage engine to implement *transactional ordered maps*. Yesquel adopts this choice. Roughly, each ordered map stores a SQL table or index, with efficient operations to insert, lookup, delete, and enumerate items in order. In addition, the engine supports transactions that span many ordered maps.

To scale, Yesquel's storage engine, YDBT, 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 neighboring 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 rebalancing 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 provide locality, YDBT arranges the data as a tree where consecutive keys are stored in the leaves. To minimize round trips, YDBT combines 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 a 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.

Other important features of YDBT are the following. First, YDBT offers strong consistency and transactions. This is expected from the storage engine of a database system. Second, YDBT can provide fault tolerance by replicating the nodes of the tree across servers. Third, YDBT is latency efficient: insert, lookup, delete, and enumeration operations often take one round trip to a server.

Technically, YDBT is a type of distributed balanced tree (DBT), which is the generic name of a distributed data structure based on a tree whose nodes are spread over many machines. We are not the first to propose a DBT, but prior DBTs [2, 42, 61] are ill-suited for our setting: some of them do not provide the required functionality (e.g., transactions, snapshots), and none scale well. We delve into these points when discussing related work (§8).

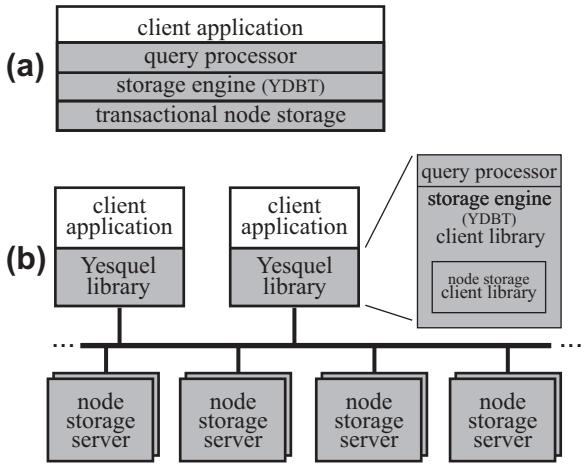


Figure 2—Logical (a) and physical (b) architecture of Yesquel, with its components shown in gray. Each client application has its own query processor as part of the Yesquel library. The query processor transforms SQL queries into operations on tables and indexes in the storage engine. At the core of the storage engine is YDBT, a distributed balanced tree which stores its nodes on a transactional node storage system, which is distributed across many storage servers.

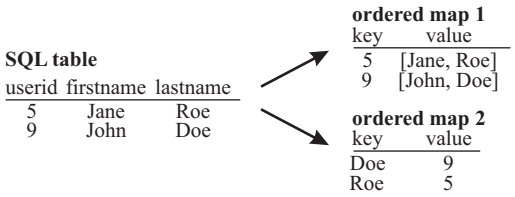


Figure 3—Example showing how a SQL table with primary key userid and a secondary index on lastname is stored as two ordered maps: one maps the primary key to the rest of the row, the other maps the secondary index to the primary key.

3. Architecture

Figure 2 shows the architecture of Yesquel. Each Yesquel client has its own SQL query processor, to which it submits SQL queries. The query processor is part of a library that is linked to the client application.

The query processor takes a SQL query and issues operations on ordered maps. The ordered maps are an abstraction provided by the storage engine. They maintain a map from *keys* to *values*, with efficient operations to search values by key and to enumerate the keys in order. Figure 3 shows how ordered maps store SQL tables and indexes.

To store the ordered map, the Yesquel storage engine uses YDBT, which consists of a tree whose nodes are spread over a set of storage servers. Leaf nodes store keys and values (or pointers to values if values are large); interior nodes store keys and pointers to other nodes.

To store tree nodes, YDBT uses a distributed storage system with multi-version distributed transactions and mostly commutative operations, which provides highly concurrent

Operation	Description
[Transactional operations]	
start()	start a transaction
rollback/commit()	rollback or try to commit transaction
[Traversal operations]	
createit()	create an iterator, return *it*
seek(it, ix, k)	move *it* to *k* in *ix*
first/last(it, ix)	move *it* to smallest or largest key in *ix*
next/prev(it)	move *it* to next or prev key
[Data operations]	
createix(db)	create index in *db*, return *ix*
destroyix(ix)	destroy index
insert(ix, k, v)	insert (k, v) in *ix*
delete(ix, k)	delete $(k, *)$ from *ix*
deref(it)	dereference *it*, return (k, v)

Figure 4—Interface to the YDBT distributed data structure, where *k* is a key, *v* a value, *ix* an index, and *it* an iterator.

access. Fault tolerance and availability are provided at each storage server, by optionally logging updates in stable storage (disk, flash, battery-backed RAM, etc.) and replicating state using known techniques.

This architecture has three new features:

- Each client has a dedicated SQL query processor. Therefore, as the number of SQL clients increases, the SQL processing capacity also increases.
- The storage engine is shared among a large number of query processors—as many as the number of clients. Thus, it must be efficient under a highly concurrent load.
- Distributed transactions are provided at the lowest logical layer (node storage), thus inverting the order of the DBT and the transactional layer in other architectures [37]. This frees the distributed data structure, YDBT, from concerns of concurrency control, improving efficiency and simplifying its design (it can use the distributed transactions of the node storage layer).

This architecture could perform poorly in workloads outside Yesquel's target of Web applications. A data analytics workload, for example, has long-lived queries that touch a lot of data (e.g., computing an average over all data). In Yesquel, such queries would move all data from storage servers to the query processors. A better architectural choice for this case would be to ship the computation to the data (e.g., computing partial averages at each server).

Client API. Figure 4 shows the interface exposed by YDBT. Basically, there are operations to store, retrieve, and delete key-value pairs; to enumerate keys in order using iterators; and to start, commit, and abort transactions.

More precisely, there are three classes of operations. *Transactional* operations start and end transactions. *Traversal* operations manipulate ordered iterators. *Data* operations read and write data.

When a transaction starts, it sets a transactional context for subsequent operations. To support many outstand-

ing transactions per process, there is an extended interface with the transactional context as a parameter to each operation (not shown). Each index belongs to a database, which is an organizational unit that groups the data of an application. Note that we distinguish "database" and "database system": the former is a data collection while the latter is a system such as Yesquel.

The operation to create an index takes a database identifier *db* and returns a fresh index id *ix*. Operations to insert and delete keys take an index *ix* and a key *k*.

Iterators provide ordered enumeration of the keys, and they can be dereferenced to obtain the key that they point to. Seeking an iterator to a nonexistent key is allowed; it sets the iterator to the previous existing key or to the end if there are none. This is useful for range queries, which seek the iterator to the beginning of the range and advance it until the end of the range.

4. The YDBT distributed balanced tree

YDBT is a key technical contribution of this paper. It is a distributed data structure with efficient operations for insertion, lookup, deletes, ordered enumeration, and transactions (Figure 4). It is used for storing Yesquel tables and indexes, one per YDBT instance.

The design of YDBT was initially inspired by B+trees, but deviates from B+trees in many ways. A B+tree is a tree data structure that implements a *key*-to-*value* ordered map; it stores all keys and values at leaf nodes, each of which stores a contiguous key interval. YDBT spreads the nodes of the tree across many servers, where a node is identified by a nodeid with a tree id, a server id, and a local object id. Pointers in the tree refer to nodeids.

Distributed B+trees have been proposed [2, 42, 61], but YDBT differs from them to improve performance and scalability: it has a new caching mechanism (§4.1), it performs splits differently (§4.2), it has new mechanisms to handle concurrent access (§4.4, §4.5), it supports snapshots (§4.7), and it uses a decentralized allocation scheme to obtain unique names for its nodes (§4.8).

Because YDBT spreads its nodes over the network, the clients of YDBT—the many SQL processors in Yesquel—need to *fetch* tree nodes across the network. (Henceforth, we use "fetch" to mean reading a node remotely from the server that stores it.) There are three immediate challenges: latency due to network round trips, load balancing as servers get overloaded, and concurrent access as multiple clients modify the tree simultaneously, or some clients search the tree while others modify it.

The design of YDBT is guided by two principles:

- *Drop cache coherence: speculate and validate.* Clients of YDBT will cache tree nodes without coherence, then execute speculatively and validate the results before taking irrevocable actions. This idea makes sense when enough of a client's cache remains valid over time.

```
procedure search-tree(k)
  n ←well-known nodeid of root node
  node ← read-cache-or-fetch(n)
  while (node not leaf and k ∈ node.fence)        // search in cache
    push(n)
    n ← find-child(node,k)
    node ← read-cache-or-fetch(n)
  while (k ∉ node.fence)                          // back phase
    n ← pop()
    node ← fetch(n)
  while (node is not leaf node)                   // down phase
    n ← find-child(node,k)
    node ← fetch(n)
```

Figure 5—Back-down search algorithm of YDBT.

- *Eliminate contention.* Use structures and operations that allow many clients to execute without creating conflicting accesses.

4.1 Caching and back-down searches

Searching a tree is the operation of finding the leaf node that stores a given key. This operation forms the basis for all others (lookup, insertion, deletion, etc.).

To search efficiently, clients have a cache of inner nodes of the tree. Because the nodes have a large fan-out, the tree has relatively few inner nodes, so clients can cache all of them even for large trees [2]. Therefore, clients can search the tree almost entirely in its cache, with the exception of the final leaf, which it fetches. A client's cache becomes stale as other clients modify the tree; thus, when the client searches the tree using its cache, it may fetch a wrong leaf. This problem is detected by having the client validate the leaf that it fetched using range fields[59]/fence intervals [22, 61]. Specifically, all tree nodes—leaves and inner nodes—have an interval of keys (the fence interval) that they are responsible for; this interval appears as a field in the node itself. When a client fetches a leaf, it checks that the key that it is searching for falls within the fence. If it does, the client is done; otherwise, the client knows that something is wrong.

The innovation in YDBT enters in this latter case; the client uses a new technique, which we call *back-down search*. Consider the path that the client took from the root to the incorrect leaf, during the search that used its cache; this path often has a large prefix that remains valid, because upper tree nodes rarely change. With back-down searches, the client proceeds in two phases. In the *back* phase, the client backtracks from the search it did in its cache, until it finds a node whose fence has the key. In the *down* phase, the client moves down in the tree from that node, until it reaches the leaf. Figure 5 shows the detailed algorithm. Under this scheme, clients search the tree with just one fetch most of the time. The algorithm ensures that, upon termination, the client has reached the correct leaf. But, the client cache may not be a well-formed tree as nodes get updated at different times. To avoid looping forever, the client limits the cache

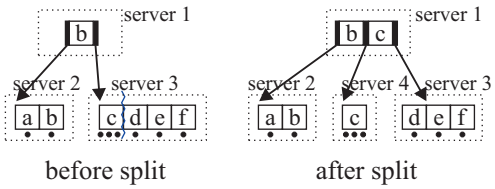


Figure 6—A tree before and after a split. The small circles indicate the load on each key. In a traditional B+tree, splits occur when a node is too large. In YDBT, splits also occur when the load on a node is too high. If the high load is caused by a single key, YDBT could employ a *replit* to split the key into two replicas (not shown).

search to 100 iterations and then invalidates the cache. This happens rarely.

The fence interval of a node can be approximated by the interval from the smallest to the largest key in the node. YDBT currently uses this approximation, which saves the space required to store the fences. The back-down search algorithm still works, but less efficiently: it might unnecessarily move back from the correct leaf only to descend again to the same node. This happens when the searched key belongs to the fence interval but not to the approximated interval. This is infrequent, except in one case: when adding the largest (or smallest) key in the DBT. In this case, we use the fact that the leaf that will store the key is the rightmost (leftmost) leaf, and therefore has an empty right (left) sibling pointer. This pointer indicates that the true fence interval extends until infinity (minus infinity). Thus, if the algorithm arrives at this node with the largest (smallest key), it need not move back.

4.2 Load splits and replits

A traditional B+tree balances itself by splitting nodes that are too large (Figure 6). This idea is designed to pack nodes into disk pages by ensuring nodes are never too large, while bounding the depth of the tree logarithmically. In YDBT, the design considerations are different. We are less concerned about large nodes than about nodes with too much load: a skewed workload might cause most accesses to hit a small number of leaf nodes, creating load imbalance across the storage servers.

To address this problem, we conceptually separate the policy and mechanism implicit in B+tree splits, and then we choose them to suit our requirements. For policy, YDBT introduces *load splits*, which splits nodes with large access rates. For mechanism, YDBT divides the node so that each new node receives (approximately) the same load; YDBT also introduces a mechanism called *replits*, which intuitively splits the load of a single key across many servers. As in B+trees, YDBT also splits nodes that are too large, creating new equally sized nodes.

To perform load splits, each server keep track of the accesses in the past second, recording the key, tree node, and operation. This is not much information even under hundreds of thousands operations per second. After each second, the server checks if the access rate of a node exceeds a threshold, in which case the system splits the node in a way that the each new node gets approximately half of the load. This calculation requires determining a split key k such that approximately half of the load of the node goes to keys less than k, and half goes to keys k or above. Using one-second intervals is admittedly simplistic. A more sophisticated idea (not implemented) is to use the one-second measurements to derive a list of popular nodes at a larger time granularity, say tens of seconds, and split them only if they remain popular for sufficiently long.

If a single key is extremely popular, there might be no good split key because, even if the popular key were on the node by itself, its load might exceed the maximum desired load for a node. To handle this case, we can split a popular key into two replicas using a hybrid between replication and split, which we name *replit*. This is an instance of the idea of replicating to improve scalability [23], and it works well when most of the operations on the key are reads (we address update operations later). To realize this idea in a search tree, each key in the tree is appended with r bits that are usually set to 0, where r is a small number, such as 8. If a key has a large read rate and small update rate, we split the key in two as follows: we create a new key by replacing the r trailing bits randomly, and then pick a splitting key between the old and new keys. In this way, the old and new keys go to different nodes. When a client wishes to search for key k, it appends r random bits to k and searches for this modified key k'. If k' is not in the tree, recall that the search procedure returns the preceding key. Because each key has a copy with the trailing bits set to 0, this guarantees that clients find the key. Because clients pick the trailing bits randomly, if a key has been split then in expectation the read load is balanced across the replicas. This idea works even if a key is replit many times. The cost of replits is that an update must change all replicas.

If the load on a popular key is due to updates, YDBT uses a different technique: it keeps a single replica, but places the key in a node by itself (singleton node). If the load on the key is higher than what the server can handle, one could place the singleton node in a more powerful machine.

4.3 Node placement

In what server are new nodes placed? The nodeid of a node determines its placement (§4.8). The system periodically samples the load at servers and allocates new nodes at the server with the least load.

In some cases, the load may be balanced across tree nodes, but a server may be overloaded due to having too many nodes. This problem is resolved by moving some of its nodes to a new server, which requires (a) copying the content of the node to a new node with an appropriate nodeid, and (b) updating the reference at the parent node.

4.4 Handling concurrency

As many parties read and write nodes on the tree simultaneously (searching nodes, splitting nodes, moving nodes), we need to ensure that the results of operations are correct and that the tree is not corrupted. YDBT adopts an architectural solution to this problem: First, build a transactional system to store tree nodes (§4.6). Then, use the provided transactions to read and modify nodes; the transactions ensure integrity of the tree. For example, splitting a node requires removing some keys from the node, creating a new node, and adding a key to the parent node; these modifications are packaged in a transaction, which is executed atomically and isolated from other transactions. YDBT also uses storage system transactions to implement its own transactions, by packaging multiple YDBT operations within the same storage system transaction. This general approach to concurrency control in DBTs was first proposed in [2] and later used in [61]. When we implemented it, however, we found that it resulted in poor scalability (reported in Section 6.1) because of a large number of aborted transactions when many clients executed concurrently. We now explain how YDBT overcomes this problem.

4.5 Improving concurrency

YDBT introduces several techniques to eliminate conflicts when updating the tree, to reduce the number of aborted transactions and improve concurrency. Roughly, a transaction aborts when another transaction touches the same data. YDBT relies on four techniques to avoid this situation:

1. Multi-version concurrency control. Prior DBTs use optimistic concurrency control, which aborts transactions at any signs of contention. YDBT uses more advanced concurrency control (§4.6).

2. Delegated splits. In YDBT, clients delegate the coordination of the split to remote *splitter processes*; each is responsible for serializing the splits to a disjoint set of nodes. The advantage of this approach, versus having the clients coordinate the splits, is twofold. First, per node, there is a single splitter, so the problem of concurrent, mutually inhibiting splits disappear. Second, because splitting is done separately, insert operations can return before the split, thereby improving latency. Currently, the splitter process responsible for a node runs at the storage server of that node, which has the advantage that the splitter can access the node locally.

3. Mostly commutative operations. YDBT uses mostly commutative operations to modify a node, so that two parties can often modify the same node concurrently without conflicts (hence without aborting their transactions). To do so, the node content is treated as an ordered list of keys, with operations to add an element e and delete an interval i; provided that i does not cover e, the operations commute. Also commutative are operations that modify the values of different keys in the same leaf node.

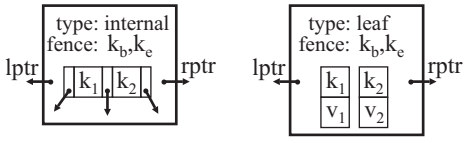
4. Right splits. There are two ways to split a node: keep the first half of keys and move the second half to a new node (we call this "left split"); or keep the second half and move the first half ("right split"). B-trees sometimes use left splits because it is easier to delete the end of an array rather than the beginning. We observe, however, that right splits are better to avoid contention in Yesquel. Specifically, keys are often added to a tree in increasing order, especially in tables with autoincrement columns—a popular SQL feature that automatically picks keys with unique increasing values. These inserts land at the tail end of the rightmost leaf node. By using a right split, YDBT ensures that the tail remains in the same node, allowing inserts to occur concurrently with a node split, using the third technique above.

Operation	Description
[Transactional operations]	
start()	start a transaction
rollback()/commit()	rollback or try to commit
[Node data operations]	
insert(n,k,v,dir)	insert (k,v) into n
lookup(n,k)	lookup k in n, return value/pointer
delrange(n,k_1,k_2,dir)	delete keys in $[k_1,k_2]$ from n
setattr(n,at,v)/*getattr*(n,at)	set or get attribute at in n
[Whole node operations]	
read(n)	read n
create$(n,type,ks,vs,ats)$	create and populate node n
delete(n)	delete node n

Figure 7—API for the transactional node storage, where n is a nodeid (128 bits), k is a key, v is a value or pointer, *dir* is "left" or "right" (indicating where an inserted or deleted pointer is situated relative to the key), *at* is an attribute id, *type* is "inner" or "leaf", *ks* is a list of keys, *vs* is a list of values or pointers, *ats* is a list of attribute ids and values.

4.6 Multi-version distributed transactional storage

YDBT stores tree nodes in a transactional storage system that shards the nodes across a set of storage servers. These servers can store data in stable storage for durability, and they can be replicated for high availability using standard techniques such as Paxos [35] or primary-backup [6]. Each server manages a log and local storage for tree nodes, where each node consists of a type, keys, values or pointers to other nodes, sibling pointers, and fence keys:



The clients of this system are (1) the YDBT operations executed by the Yesquel query processors, and (2) the splitter processes. Clients have a simple API to read and manipulate the nodes (Figure 7).

The key design choices of the storage system are these:

Provide fine-grained operations. Many operations in the API modify a small part of a node's state, ensuring that opera-

tions mostly commute with each other to minimize aborts of transactions (§4.5).

Use multi-version concurrency control (MVCC). Transactions can be implemented in many ways. We use multi-version concurrency control (MVCC) [8], which manages multiple versions of data to improve concurrency. In particular, read-only transactions, the most common type, never need to lock any data and never abort. The trade-off is larger consumption of storage, since the system keeps many versions of each node. This trade-off is reasonable because (a) storage is cheap, and (b) old versions need to be kept only for the duration of transactions, which last for less than a second (see Section 2.1).

Run commit at the clients. Latency is key, and so clients themselves run the commit protocol instead of delegating to an external coordinator, as in some distributed systems. Doing this complicates recovery from crashes, but this is the right trade-off since crashes are relatively infrequent. To recover, we use the protocol in Sinfonia [5, Section 4.3]. Its key idea is that the transaction outcome is a sole function of the votes: the transaction has committed iff all participants have voted yes. Since the coordinator state is irrelevant for determining the transaction outcome, the system can recover without the coordinator (client). This is done by a recovery process that periodically visits the servers to check on pending transactions.

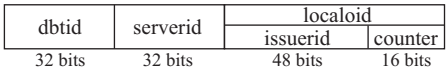
Use clocks, but not for safety. Clocks can efficiently order events in a distributed system. But clocks are unreliable: time drifts even under synchronization protocols such as NTP (e.g., due to errors in the client configuration or in firewall rules), or time may move backwards *due to* the synchronization protocol. Fortunately, these problems are rare; most of the time, clocks are correct (e.g., within milliseconds or less of real time). It is therefore desirable to use protocols that can use clocks when they are correct, but that do not violate safety when they misbehave. We use a transactional protocol similar to the ones in [15, 66], which combine many existing ideas: two-phase commit [24], two-phase locking [17], timestamp-ordering [8, 56], and mixed multi-version schemes [9]. We omit the details for brevity's sake.

4.7 Snapshots

A snapshot refers to the state of the data (the nodes of all trees) at a given moment. Snapshots are immutable by definition and they are useful to support iterators when data may change. YDBT obtains snapshots for free from its transactional storage, which ensures that each transaction reads from a snapshot (§4.6). Compared to the approach in [61], in which the DBT directly implements snapshots, YDBT is simpler and faster (§6.1.6). This is a case where, when functionality is provided at the right layer (the low-level storage layer rather than the data structure layer), it becomes both simpler and more efficient.

4.8 Choosing nodeids

Each tree node is identified by a 128-bit nodeid, which is used in the storage API (Figure 7) and serves as a pointer to the node. The higher 64 bits of the nodeid indicate the tree and server to which the node belongs.



The lower 64 bits are the *localoid*. The root node has localoid 0. For other nodes, the localoid is divided in two parts: 48 bits for an issuerid and 16 bits for a monotonic counter. An issuer is a process that wants to allocate new nodes (e.g., the splitter processes); each issuer is assigned an issuerid and creates new localoids *locally*, by incrementing the 16-bit counter. This scheme can support $2^{48}\approx$281 trillion issuers and 2^{16}=64*K* localoids per issuer. Should an issuer need more localoids, it obtains a new issuerid to get another 64K localoids. There is no need to recycle issuerids or localoids after deletion: with 48 bits, there are enough ids to serve 5K new issuers per second for 1700 years.

When an issuer starts, it contacts a management server to obtain a new issuerid; the server issues issuerids using a monotonic counter that starts with 1. To ensure monotonicity despite failures, the management server stores the last issued issuerid in a special node in the storage system. This needs to be done before the server returns the id to the issuer, but it can be batched: if many issuers ask for ids in a short period, the server can store only the value of the largest id before returning to the issuers.

5. Implementation details and optimizations

Excluding its query processor, Yesquel is implemented in 33K lines of C++, where roughly a third is the DBT, a third is the node transactional storage system, and a third is general-purpose modules (data structures, RPC library, etc.). Yesquel uses the query processor of SQLITE [62] version 3.7.6.3. SQLITE is an embedded database system used in many commercial applications.

Optimizations. SQLITE's query processor is designed for local storage and produces inefficient access patterns for remote storage. We implemented several optimizations in Yesquel to reduce the number of RPCs that a client must issue. Some of them are the following:

- *Per-transaction cache.* The Yesquel query processor often reads the same data multiple times in the same transaction. Therefore, we implemented a local-to-the-transaction cache within the client library of the node storage system (§4.6). Note that this cache does not affect consistency because a transaction reads from a snapshot.
- *Reading keys without values.* A DBT leaf stores keys and values, but values can be very large. To save network bandwidth, when performing a seek the client library

reads a leaf node without its values. Later, it can request the specific value that it wants.

- *Deferred seek.* The Yesquel query processor sometimes seeks an iterator with the sole intention of dereferencing it to obtain a key-value pair. The previous optimization creates an inefficiency in this case: seeking an iterator causes an RPC to fetch a leaf node without its values, while dereferencing the iterator causes another RPC to fetch the desired value. To save an RPC, another optimization defers execution of the seek until the iterator is dereferenced or moved. If the iterator is dereferenced, YDBT fetches the required value from the leaf without the keys. If the iterator is instead moved, YDBT fetches the keys from the leaf without the values.
- *Deferred write.* The node storage system buffers small writes of a transaction at the client, until the transaction commits. Later, the writes are piggybacked onto the prepare phase of the commit protocol. This optimization saves RPCs on writes.
- *Optimistic insert.* Without this optimization, inserting an element into a DBT is performed using a back-down search (§4.1) to fetch the appropriate leaf node and then insert the element into that leaf node. This takes two RPCs. The optimization avoids fetching the leaf node by embedding the insert into the back-down search algorithm. Specifically, the client first searches in its cache of inner nodes to find a probable leaf node. Then, rather than fetching the leaf, the client optimistically requests the storage server to insert the element into that node, which costs one RPC (if successful). The server's fence interval prevents insertion into an incorrect node.

Design differences. Yesquel's implementation differs from its design in one significant way: the implementation does not yet replicate storage servers (using Paxos, as discussed in Section 4.6) or keys (using replits, as discussed in Section 4.2). This is future work.

6. Evaluation

Our evaluation proceeds in two stages. First, we take Yesquel's architecture as a given and inspect a key component:

- How well does YDBT perform, and how effective are its new techniques? (§6.1)

Next, we compare architectures. We assess whether Yesquel meets its top-level goal of providing the features of SQL with performance and scalability akin to NOSQL systems:

- Is Yesquel competitive with NOSQL key-value systems on workloads designed for them? (§6.2)
- Is Yesquel competitive, in terms of functionality and performance, with existing SQL systems on the SQL queries of a real Web application? Does it scale? (§6.3)

Figure 8 summarizes this section's results.

• YDBT performs and scales well, significantly outpacing prior DBTs; its new techniques enable this performance.	§6.1
• Yesquel's scalability, single-server throughput, and latency are almost as good that of Redis (a commonly used NOSQL system) under reads, updates, and scans.	§6.2
• While offering the same functionality as MYSQL (a popular SQL database system), Yesquel has significantly better single-server throughput and latency.	§6.2.1 §6.3.1
• Yesquel has significantly better scalability than MYSQL-NDB (a distributed database system).	§6.2.2
• Compared to the aforementioned baselines, Yesquel's principal resource cost is additional network bandwidth, memory, and CPU cycles at clients.	§6.2.1 §6.3.1
• Given the SQL workload of the Wikipedia Web app, Yesquel provides good latency and near-linear scalability.	§6.3

Figure 8—Summary of evaluation results.

Testbed. Our testbed consists of a shared cloud environment with 144 virtual machines (VMs), each with one virtual CPU, running Linux (v3.16.0-30). Each VM has 6 GB of RAM. There are 16 physical hosts, each running up to 10 VMs. We ensure that communicating VMs are not placed in the same physical host, so that communication crosses the physical network. The physical hosts were reserved for our exclusive use (we first shared the physical hosts among cloud users but this led to unrepeatable measurements.) A physical host has two 10 Gb network adapters, and the ping round-trip latency between VMs in different hosts is 0.14 ms. Throughout this section, "machine" refers to a VM.

6.1 Evaluation of YDBT

This section compares YDBT against prior DBTs that offer the same functionality. Our experiments present YDBT with various workloads (listed in Figure 9); some are chosen to isolate YDBT's individual techniques and others to measure its overall ability to handle concurrency and to scale.

Baselines. The baselines are the previous transactional DBTs in the literature: the Sinfonia DBT [2] and Minuet [61]. Source code is unavailable, so we provide our own implementation, using YDBT's code base:

DBT	description
BASE	Represents [2, 61]. Optimistic concurrency control (instead of multi-version concurrency control as in YDBT); no back-down searches, load splits, or delegated splits.
BASE+	Adds YDBT's back-down searches to BASE

Setup. We experiment with different workloads, DBTs (the two baselines and YDBT), number of clients, and number of servers. We almost always assign two clients per server, each with 32 threads in closed loop (there are two exceptions noted later). Each thread issues requests, according to the given workload, to the DBT client library. Servers operate in memory only, with disk persistence disabled. Our princi-

workload	description	models what in SQL?
Read, Read-2 (§6.1.1–§6.1.3)	10k keys in DBT (200 keys in DBT for Read-2). Read randomly chosen keys.	SELECT statements (SQL row or index read causes DBT read)
*Update (§6.1.1, §6.1.2)	10k keys in DBT. Change values for random keys.	UPDATE statements
*Insert (§6.1.1)	10k keys in DBT to begin. Insert new random keys.	INSERT statements
*Insert-mono (§6.1.4)	Same as prior, but new keys inserted in increasing order.	Autoincrement (§4.5). Also, loading a table in key order.
*Scan-up-tx (§6.1.5)	10k keys in DBT. DBT transaction: scan n items (at random starting point), then update on a random key.	Explicit SQL transactions and multi-key operations (e.g., store result computed from many keys).
*Update-scan (§6.1.6)	10^6 keys in DBT. Update on random keys; a scan runs concurrently.	scans come from SELECTs with range queries, some joins, enumerations.

Figure 9—Workloads for DBT experiments; asterisk indicates transactional use of the DBT. Keys always have 64 bits; values have 100 bytes unless stated otherwise in the text.

	update/read		insert	
# clients	YDBT	BASE	YDBT	BASE
1	0%	0%	0%	0%
5	0%	0%	0.02%	5.1%
10	0%	0%	0.03%	10.8%
20	0%	0%	0.06%	20.3%
30	0%	0%	0.10%	28.0%
40	0%	0%	0.13%	34.5%
50	0%	0%	0.16%	39.8%

Figure 10—Fraction of operations that access the root node as we vary the number of clients. Numbers for BASE+ are identical to YDBT in all cases (omitted).

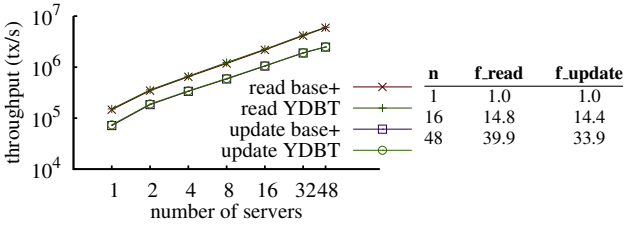
pal measurement is throughput of successful or committed operations, taken over one-second intervals.

Standard deviations are always under 5% of the mean. Throughout the paper, ± in tables indicate the standard deviations; graphs omit error bars.

6.1.1 Root node load

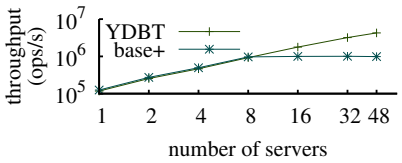
A key challenge for a DBT is reducing load on the root node. We evaluate YDBT's response to this challenge with the Read, Update, and Insert workloads. In each experiment, we fix a DBT and number of clients, and we measure the number of accesses to the root for 5 million operations.

Figure 10 depicts the results. For the Read and Update workloads, YDBT and BASE both avoid accesses to the root, YDBT by using back-down searches (§4.1) and BASE by replicating its version numbers [2]. For the Insert workload, YDBT's back-down searches ensure that clients access the root only when it is modified, which happens rarely. By contrast, in BASE, as the number of clients increases, an increasing fraction of inserts access the root (up to ≈40% in our experiments). The reason is that when a node splits—



n	f_read	f_update
1	1.0	1.0
16	14.8	14.4
48	39.9	33.9

Figure 11—Read and update performance of YDBT and BASE+. For reads and updates, performance coincides in both systems and scales well. With one server, performance of YDBT is 152K reads/s and 74K updates/s. The table shows the scalability factor f for n servers (improvement over a one-server system) for reads and updates on either YDBT or BASE+.



n	f_ydbt	f_base+
1	1.0	1.0
8	7.9	7.8
32	27.3	8.0
48	36.3	7.9

Figure 12—Small-tree read performance of YDBT (with load splits, §4.2) and BASE+ (with size splits).

which happens often with inserts—clients eventually trip on it, owing to outdated caches, which requires restarting the search by reading from the root. Thus, each split ultimately causes ≈c reads of the root, where c is the number of clients. This ultimately overloads the root, presenting a scalability bottleneck. Thus, to study other scalability issues below, we augment BASE with back-down searches—this is BASE+.

6.1.2 Reads and updates

We next consider the one-server performance and scalability of read and update operations. These are operations for which prior DBTs perform and scale well (linearly, in fact [2, 61]), and we investigate whether YDBT can match that. These experiments also establish the base overall performance of our testbed. Specifically, we run the Read and Update workloads, modified to use 64-bit data values, which are common for database indexes. Each experiment runs for two minutes; we discard the first and final 15 seconds (to ensure that all clients participate in the measurement interval despite possibly starting at slightly different times).

Figure 11 depicts the results. We can see that BASE+ and YDBT are similar and fast, executing 10–100K op/s on a single server, and that performance scales well.

6.1.3 Load splits

We measure the effects of YDBT's technique of load splits (§4.2). We compare YDBT to BASE+, which uses the standard technique of *size splits* (§4.2), configured for 50 keys per node. We use the Read-2 workload (which concentrates load), varying the number of servers. Each experiment runs for two minutes, and we discard the first and final 15 seconds.

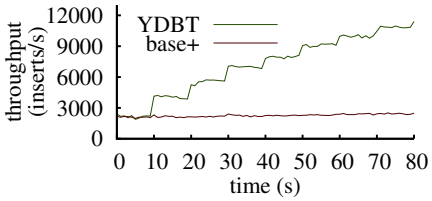


Figure 13—Effect of contention on throughput of increasing-key inserts as more clients execute, in Yesquel and BASE+.

Figure 12 shows the results. With load splits, the DBT scales up to 48 servers (the maximum we experimented with). With size splits, the DBT stops scaling at 8 servers. The reason is that the tree has few leaves and cannot use the additional servers; indeed, with 48 servers, we find that most are idle. To be sure that we have isolated the effect of load splits, we experimented with a version of YDBT that has BASE+'s split mechanism. This alternative performs identically to BASE+ under this workload.

We have not done this experiment with an update-heavy workload, but we expect the same result: under size splits, servers without tree nodes will be idle, impairing scalability.

6.1.4 Insert contention

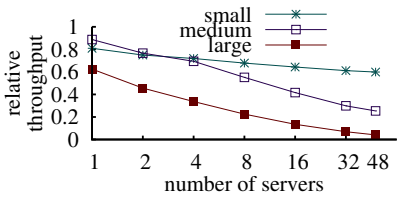
We assess YDBT's ability to handle concurrent conflicting accesses to the same node, as results from SQL's popular *autoincrement* columns (§4.5). We use the Insert-mono workload and 16 servers. At the beginning, a single client with one thread inserts increasing keys in a closed loop; every 10 seconds a new client with one thread joins the system doing the same. The inserted keys are handed out by a single server so that each client inserts the largest key thus far.

Figure 13 depicts the time series of throughput in our experiments. Performance does not scale in BASE+, because of conflicting operations on the rightmost leaf of the DBT, causing frequent transaction aborts. In contrast, YDBT scales well. We also run the experiment with various features of YDBT individually disabled (delegated splits, mostly commutative ops, right splits). We observe that throughput drops significantly without these techniques, up to 80% or 5× slower (due to thrashing caused by aborted transactions):

technique disabled	performance drop
delegated splits	55.0%
mostly commutative ops	80.0%
right splits	80.7%

6.1.5 Multi-key transactions

We assess the union of the techniques in Sections 4.1–4.6, by investigating the two DBTs' ability to handle multiple-key transactions of different sizes at different system scales. We use the Scan-up-tx workload, configured as *small*, *medium*, *large* (corresponding to the number of items retrieved by the scan portion of the transaction: 2, 10, 100) and vary the number of servers. Each experiment runs for two minutes; we discard the first and final 15 seconds.



	1 server		48 servers	
transaction size	**YDBT**	**BASE+**	**YDBT**	**BASE+**
small	28.9	23.4	625.5	374.5
medium	11.0	9.8	347.9	88.3
large	1.5	0.93	54.7	2.2

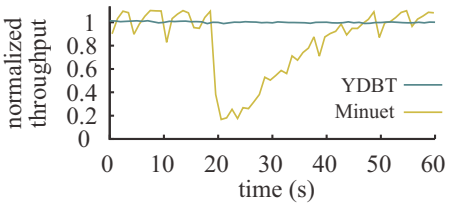
Table shows thousands transactions per second

Figure 14—Throughput of YDBT and BASE+ for small, medium, and large multi-key transactions. The graph shows the relative throughput of BASE+ over YDBT.

Figure 14 depicts the results. BASE+'s performance drops by 11–38% with one server and 40–96% with 48 servers. The reason is that BASE+ uses optimistic concurrency control, which has two drawbacks: (1) when each transaction completes, the system must check to see if the scanned items have changed, which incurs additional RPCs, and (2) the transaction aborts if another transaction writes to any of the scanned items. Both drawbacks intensify with larger scans. The second drawback also intensifies with larger system sizes, which in our setup has more clients, increasing the conflicts between update and scan of different transactions.

6.1.6 Snapshots

What is the effect of snapshotting (§4.7) on the performance of concurrent updates? To answer this question, we run the Update-scan workload with 16 servers and 32 clients issuing updates. After 20 seconds, a single client thread begins a long scan operation (which implies a snapshot) to retrieve 10% of all keys. The time series of this experiment is below, where the y-axis is normalized to the average throughput before the scan starts (around 1M updates/s for YDBT). We overlay the graph with published results of a similar experiment with Minuet [61, Fig. 14]:[1]



The scans have no observable effect on the update throughput of YDBT; Minuet's throughput, in contrast, drops around 80%. The reason for YDBT's low overhead is that multiversioning—provided in YDBT by its transactional storage layer (§4.6)—in general yields snapshots for free.

[1] In Minuet, a scan is preceded by an explicit request to create a snapshot, whereas in YDBT, every transaction establishes a snapshot for itself.

6.2 Yesquel under key-value operations

This section investigates whether Yesquel matches the performance and scalability of NOSQL systems. We compare the two under workloads of simple key-value operations. We also include SQL database systems in this study. This is for completeness: if the SQL database systems could perform similarly to NOSQL systems—which we do not expect to happen—then there would be little or no role for Yesquel. We separately consider single-server performance and scalability. The former establishes a benchmark for judging the latter; also, if the former is poor, then the latter is moot.

Baselines. The baselines are Redis (v2.8.4), a NOSQL key-value storage system; MYSQL (v5.5.44), a central database system; and MYSQL-NDB (v7.4.7), a distributed database system. Redis is a very popular NOSQL system; it is used in many Web sites (Twitter, Pinterest, Instagram, StackOverflow, Craigslist, etc.). MYSQL is also popular, but it is limited to a single server. MYSQL-NDB consists of the MYSQL front-end with the NDB distributed storage engine. We configure all systems to operate in memory only: for Redis we disable persistence, for MYSQL we use its memory storage engine, and for MYSQL-NDB we enable the diskless option.

Redis servers are set up to replicate the entire dataset; one server is the primary, the others are backups. A client connects to one Redis server, where it executes all its operations.

MYSQL-NDB is set up to run a MYSQL front-end (MYSQLD) at each client, which connects to MYSQLD locally, while each storage server runs a storage engine daemon (NDBD).

Workloads. All workloads presume an initial dataset with one million key-value pairs or rows, each with a 64-bit key and a 100-byte value. We consider three workloads:

1. *Read* reads the value of a varying key k, each read operation accessing a key uniformly at random.
2. *Update* transactionally reads the value of key k, again selected uniformly at random, and then writes key k with another value;
3. *Scan* has a 95-5 mix of scan and update operations; each scan operation is a transaction that enumerates n (chosen randomly from 1…5) consecutive key-value pairs, starting from an existing (random) key k.

These workloads exercise common operations in a NOSQL key-value storage system. We have optimized the current Yesquel prototype to handle these workloads, by minimizing the number of RPCs that Yesquel generates to execute these operations. Of course, NOSQL systems have other operations—Redis, for example, implements many data structures, such as lists, sets, and bit arrays. A production version of Yesquel might optimize other simple NOSQL operations too; we explain how in Section 6.4.

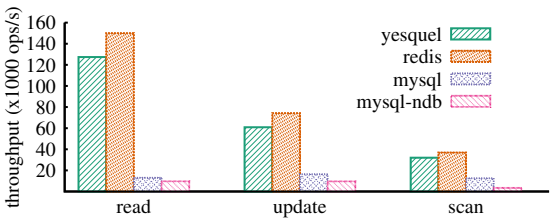
Setup. For Redis, the workload is generated by invoking the required operations using its official C interface. Redis has no indexes for ordered traversals of keys, and we cannot use Redis's scan operation because its starting key is not user-chosen. To implement scans, therefore, we manually pre-populate an index over all keys in a given experiment (without their associated values), using a sorted set; a "scan" is then just iteration over the sorted set.

For Yesquel, MYSQL, and MYSQL-NDB, clients execute the workload operations using the following SQL statements:

read(k)	SELECT value FROM table WHERE key=k
update(k)	UPDATE table SET value=f(value) WHERE key=k
scan(k,n)	SELECT value FROM table WHERE key $\geq k$ LIMIT n

These statements execute transactionally, which is pessimistic for the performance of the SQL systems and Yesquel.

We always assign two clients to a server; each client runs 32 threads in a closed loop. Experiments run for two minutes, and we discard the first and final 15 seconds. For throughput, we record measurements over 1-second intervals, and we report averages and standard deviations over these measurements. For latency, we record histograms with buckets of 0.1ms granularity, and report median and 99-percentile values. For resource consumption, we measure memory using top, and network bandwidth using ifstat, both with 1-second granularity, and we compute averages across 1-second periods. For network bandwidth, we add inbound and outbound bandwidth and divide by operation throughput to obtain bytes per operation. For CPU consumption, we compute the difference of CPU time between the beginning and end of the measurement period, and we divide by the number of operations to obtain CPU time per operation. For all client resources, we report averages across clients.



throughput (x1000 ops/s)

workload	Yesquel	Redis	MYSQL	MYSQL-NDB
read	127 ± 0.4	150 ± 0.3	13 ± 0.0	9.8 ± 0.0
update	61 ± 0.2	74 ± 0.2	16 ± 0.1	9.7 ± 0.1
scan	32 ± 0.1	37 ± 0.1	13 ± 0.0	3.4 ± 0.0

50th (99th) percentile latency (ms)

workload	Yesquel	Redis	MYSQL	MYSQL-NDB
read	0 (0)	0 (0)	4 (8)	5 (9)
update	1 (1)	0 (1)	3 (7)	7 (11)
scan	2 (3)	1 (2)	4 (9)	18 (24)

Figure 15—One-server performance of Yesquel and baselines.

6.2.1 One-server performance

Figure 15 reports performance and Figure 16 reports resource costs; we omit read costs because they are similar. We can see that Yesquel performs almost as well as Re-

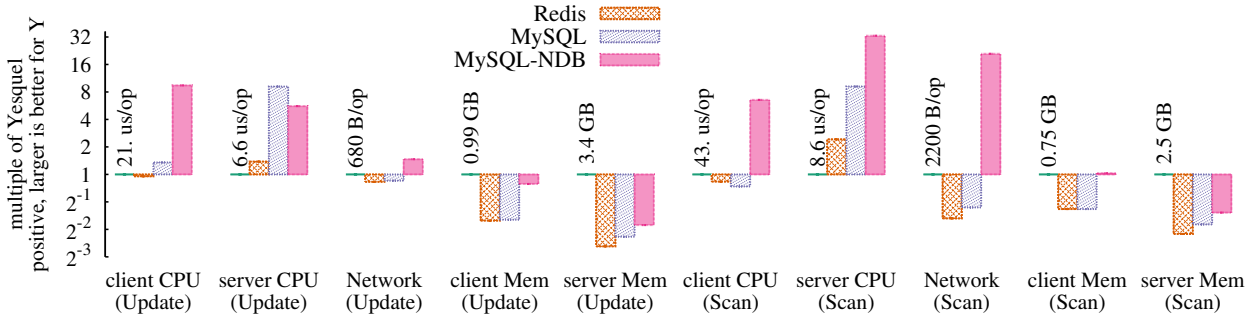


Figure 16—Resource costs—CPU (μs/op), Network (Bytes/op), Memory (GB)—for single-server experiments under scan and update. Bars are normalized to Yesquel and plotted on a log scale; labels indicate absolute data for Yesquel. Positive bars means higher resource consumption by the other systems; negative bars mean the reverse. For example, the others use less memory than Yesquel.

dis: its throughput is at most 18% lower for any workload. In absolute numbers, Yesquel performs at 127 Kops/s, 61 Kops/s, and 32 Kops/s for reads, updates, and scans, with a 99-percentile latency of a few milliseconds. Redis performs better than Yesquel because it uses a hash table to lookup keys and keeps a single version of data. MYSQL is worse than Yesquel because (a) it processes all SQL requests at the central server, causing its CPU to saturate while clients are less loaded, and (b) its SQL processing is heavier than that of SQLITE (the query processor of Yesquel, §5). MYSQL-NDB is worse than Yesquel because it has a heavier query processor and, for each read or update, it incurs multiple round trips between client and storage server, whereas Yesquel incurs one or two round trips; for scans, MYSQL-NDB's performance reflects its mechanisms to coordinate access to shared data using coarse-grained locking.

The costs of Yesquel are the following. First, Yesquel consumes more network bandwidth than Redis or MYSQL because its clients bring data to the computation. Second, Yesquel consumes more memory than other systems because the current implementation uses large node caches at the client and keeps many data versions at the server.[2] Third, on scans, Yesquel consumes more client CPU than Redis and MYSQL. This is fundamental to Yesquel's architecture, and this cost increases with the complexity of the query (scans are significantly more complex than updates).

6.2.2 Scalability to many servers

We now examine the scalability of Yesquel as we add more servers to the system, up to 48 servers and 96 clients. MYSQL cannot run with more than one server, so we do not consider it. MYSQL-NDB supports many servers, but we cannot run it with 48 servers (and 96 clients) due to its internal limits; we thus report its numbers with up to 32 servers.

Figure 17 depicts the scalability of read, update, and scan workloads for each system. For reads and updates, the three systems scale almost linearly; for scans Redis and Yesquel scale almost linearly. For scans, MYSQL-NDB does not scale since its performance is limited by coarse-grained locking (§6.2.1). In absolute numbers, with 48 servers Yesquel performs at 5321 Kops/s, 2454 Kops/s, and 1385 Kops/s for reads, updates, and scans.

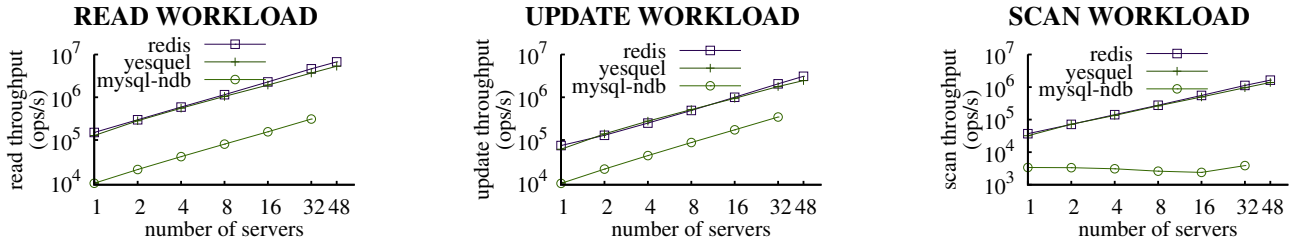
[2] It should be possible to save memory with greater limits on cache sizes and more aggressive garbage collection of versions; however, more investigation is needed to understand how this would impact performance.

6.3 Yesquel under a real Web application

For Yesquel to achieve its goal, it must handle actual SQL, scalably. We investigate whether it does so in this section, by focusing on a real Web application. We choose Wikipedia [69] because its underlying system, MediaWiki [44], already uses SQL.

Baseline. For single-server experiments, the baseline is MYSQL; for multi-server experiments, we compare Yesquel against idealized linear scalability. Wikipedia cannot run with the MYSQL memory storage engine because the engine does not support all the required SQL functionality; therefore, we use the standard MYSQL engine, InnoDB. InnoDB is designed for disk operation, but we make it run in memory by (1) disabling log flushing, (2) configuring a memory cache large enough to hold all of the Wikipedia dataset, and (3) populating the cache by loading the entire database at the beginning of the experiment.

Workload and setup. In each experiment, we load the systems with all 190K pages (430 MB) of the Simple English Wikipedia. To avoid bottlenecks elsewhere in the system (Apache, PHP), we run our experiments directly against the database system. We derive the workload by tracing the (multiple) database queries generated when a user visits one Wikipedia article. These queries populate the contents of the displayed page, which can have hundreds of KB. We modify the traced queries so that the visited article is a parameter that can be driven by a workload generator, which chooses articles to visit uniformly at random. We remove a few queries from the workload, namely queries that access tables that are empty in Wikipedia's public dataset (e.g., the user table). In the resulting workload, each Wikipedia page visit generates at least 13 SQL queries, with an additional query for each link on the page. A sample query is



scalability of Yesquel	
n	f
1	1.0
16	16.2
48	41.8

latency at max servers	50th (99th) percentile
Yesquel	0 (3)
Redis	0 (1)
MYSQL-NDB	6 (12)

scalability of Yesquel	
n	f
1	1.0
16	15.5
48	40.3

latency at max servers	50th (99th) percentile
Yesquel	1 (7)
Redis	1 (2)
MYSQL-NDB	6 (11)

scalability of Yesquel	
n	f
1	1.0
16	15.5
48	43.2

latency at max servers	50th (99th) percentile
Yesquel	2 (8)
Redis	2 (3)
MYSQL-NDB	497 (829)

Figure 17—Scalability of reads, updates, and scans. Graphs show throughput, left tables show the scalability factor f for n servers, right tables show latency at the maximum number of servers for each system.

```
SELECT * FROM page LEFT JOIN page_props ON (pp_propname =
'hiddencat' AND (pp_page = page_id)) WHERE (page_namespace = '14' AND
page_title IN ('Database', 'Structured_query_language'))
```

As before, we assign two clients to each server; each client has 32 threads querying the database in a closed loop. Experiments run for two minutes; we discard the first and final 15 seconds. We measure the throughput in terms of page views aggregated over one-second intervals.

6.3.1 One-server performance

The single-server results are the following:

	throughput (views/s)	50 percentile latency (ms)	99 percentile latency (ms)
MYSQL	913±3	68	102
Yesquel	2301±5	27	54

	memory client/server (MB)	network use per op (KB)	CPU use per op client/server (μs)
MYSQL	271/908	11	521/1091
Yesquel	1097/4030	51	646/68

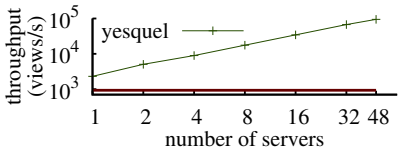
Yesquel performs well at a single server at roughly two times the performance of MYSQL. In absolute numbers, Yesquel serves 2.3K pages/s, while MYSQL serves 913 pages/s. Latency is acceptable in both systems, but Yesquel's is smaller by half. Yesquel consumes more memory due to its client cache and multi-version storage, and it consumes more network bandwidth as it moves data to the client for computation (§6.2.1). MYSQL spends significant CPU at the server, its bottleneck; Yesquel consumes more client CPU.

6.3.2 Scalability to many servers

Figure 18 depicts throughput scaling. With 48 servers, Yesquel scales to 95K pages/s, which is ≈41 times better than one server, representing near-linear scalability. Latency at the median and 99 percentile are 41 ms and 69 ms, respectively, which suits Web applications.

6.4 Discussion

Yesquel's evaluation and current prototype have some limitations. First, we have not yet run Yesquel under standard



n	f
1	1.0
4	3.9
16	14.9
48	41.2

Figure 18—Throughput and scalability factor for Wikipedia workload. The horizontal line at the bottom represents the throughput of a single MYSQL server.

SQL benchmarks, but we expect Yesquel to perform well on those for online transaction processing.

Second, we optimized Yesquel for certain common NOSQL operations (read, update, scan) and demonstrated that it performs well under those operations. Optimizing other NOSQL operations is future work. For example, a blind write (a write without a previous read) incurs one RPC in Redis and two RPCs in Yesquel: the first checks that the row exists, the second writes the row. This is due to Yesquel's query processor (from SQLITE), which is not optimized for remote storage, but it is not a fundamental limitation: the query processor could issue a single RPC that conditionally writes the row if it exists. As another example, incrementing a value in Yesquel incurs two RPCs: one reads the value, the other writes it back. In contrast, Redis increments with one RPC since it supports counters. This difference is not fundamental either: Yesquel could be augmented with server stored procedures (§7).

Third, our evaluation uses 64-bit integer primary keys. Meanwhile, Yesquel is optimized for such keys: non-integer keys incur an additional RPC on many operations. This is because SQLITE implements non-integer primary keys via a level of indirection: it maps a non-integer key to an integer key using an index, and then maps the integer key to the rest of the row through another index. Again, this is not a fundamental limitation, since the query processor could support non-integer primary keys directly.

A fundamental limitation of Yesquel is that it performs poorly on data analytics queries that operate on lots of data.

For example, computing the average access count across all Wikipedia pages takes 19 s in Yesquel and less than 10 ms in MYSQL. Such queries are better handled by a different database system design (§3).

7. Extensions

Yesquel can be extended in many ways. These extensions are common in traditional database systems, but they require additional study to be applied to Yesquel, which we leave as open directions for research.

1. Row stored procedures. Currently, Yesquel updates a row with two RPCs, one to read the row, the other to write its new value. The extension is to have procedures at the storage servers that can update a row locally, requiring a single RPC. These procedures might, for example, increment or decrement a selected column within the row.

2. Operators at the storage server. Currently, Yesquel executes all relational operators at the client. For example, to compute an average over all rows, a client reads each row. The extension is to run operators at the storage server for distributing the computation. For instance, a query processor computes the average by invoking at each server an operator that returns the count and sum of all values in a DBT.

3. Statistics and query optimization. Often, a SQL query can be executed in many ways; a query optimizer tries to predict the most efficient way based on statistics (e.g., data selectivity, cardinality, server load). Currently, Yesquel has a simple query optimizer from SQLITE, which is not designed for distributed systems. The extension is to create a better optimizer and determine the relevant statistics for Yesquel.

8. Related Work

Yesquel's architecture was broadly described before in a patent filing [4] and a keynote [3]; this paper elaborates on the architecture, describes the DBT in detail, and evaluates the performance of a prototype.

Scaling SQL database systems. We are far from the first to want to scale SQL systems. Below, we outline existing approaches to this problem.

Big Data and data analytics. There has been much work on systems that process massive data sets at many hosts (e.g., [12, 18, 26–28, 45, 54]). Some of these systems support SQL (e.g., Shark [70], AsterData [20], Facebook Presto [53], Impala [32], LinkedIn Tajo [65], MapR Drill [14]). However, they are intended for data analytics applications and are not directly applicable to serving Web applications: the queries in these two application domains have very different characteristics, as noted in Section 2.1. HANA SOE [21] supports both data analytics and online transaction processing, but scalability is limited by a centralized transaction broker. See Mohan [47] for a useful survey of Big Data systems.

Replication. Database replication is widely deployed, and most database systems provide some form of replication. Replication can scale reads, but it is less effective to scale writes [23] because adding more replicas increases the work incurred by writes proportionately. This problem can be mitigated by several techniques: lazy replication, atomic broadcast, weaker replica consistency and isolation levels, commutative operations, abstract data types, escrow methods, etc. A good overview is given in [10].

Distributed database systems. Traditionally, there have been two broad architectures for distributing and scaling a SQL database system: shared nothing and shared disk [38]. Each has advantages, and there is a long-running debate about the two approaches, with commercial vendors supporting one or the other, and sometimes both. To scale, shared nothing systems need to carefully partition data so that queries can be efficiently decomposed—but this partition may not exist or may require a (well-paid) database administrator to identify [63]. Shared disk systems need distributed locking and caching protocols for coordinating access to the shared disks; these protocols can become a scalability bottleneck (as shown in §6.2 for MYSQL Cluster).

Yesquel is a variant of shared disk. Under Yesquel, the "shared piece" is not a collection of network-attached disks but rather a distributed storage system. This picture is not unique to Yesquel; it appears in MoSQL, F1, FoundationDB, and Tell, as we discuss shortly.

NEWSQL. This is a class of SQL database systems that have adopted new architectures or ideas to improve upon traditional database systems. The class includes systems for data analytics (covered above) and for online transaction processing. The latter includes in-memory centralized systems such as Hekaton [13] and distributed systems. Since centralized systems do not scale beyond one server, we focus on distributed systems below.

H-Store/VoltDB [30, 51] is an in-memory shared-nothing system. As noted earlier, shared-nothing systems require a good data partition for good performance; software such as VoltDB's can help with this issue, but not in all cases.

MoSQL [67, 68], F1 [55, 60], FoundationDB [19], and Tell [40] implement a SQL database system atop a NOSQL storage system, following an architectural decomposition due to Deuteronomy [37]. In Deuteronomy, the storage engine is decomposed into a transactional component running *above* a data component without transactions. Yesquel's architecture reverses this idea: it layers a transactional component *below* a data component (YDBT). Doing so can simplify the system and enhance performance (§3,§4,§6.1.6).

MoSQL is mostly unoptimized: its baseline performance (reported 3× slower than MYSQL [68], which is one of our evaluation baselines) is somewhat below our performance goal. More fundamentally, MoSQL's DBT uses a different design from Yesquel's DBT; MoSQL's is based on optimistic concurrency control, and (though we cannot be sure because

the description is not detailed) appears to resemble [2, 61], whose performance is studied in Section 6.1.

F1, by contrast, is designed for performance. Its SQL query processors run over Spanner transactions [11], which run over the BigTable data component. Although F1 and Yesquel have similar high-level descriptions, they represent different design points due to different architectures (noted above) and goals. The authors of F1 indicate that it was designed for one critical application—Google's AdWords—and indeed, F1 uses application-specific techniques. In particular, F1 uses denormalization (described in Section 2.2), and it exposes a manually designed, non-relational hierarchical model. These choices contrast with Yesquel's goal of presenting an unmodified SQL interface. Finally, F1 assumes special infrastructure, such as GPS, atomic clocks, etc. (while Yesquel works even if clocks are not synchronized). On the other hand, F1 can handle short and long queries, by dividing queries into parts executed by many query processors (while Yesquel is mainly designed for short queries); and F1 provides strict serializability (while Yesquel provides snapshot isolation).

FoundationDB runs atop a transactional multi-version key-value storage system, but little is known about its storage engine and whether it uses DBTs.

Very recently, Tell implemented snapshot isolation transactions atop a key-value storage system using load-link and storage-conditional (LLSC) primitives. Indexes are stored in DBTs, which are also implemented with LLSC. Evaluating Yesquel's performance against Tell is future work.

Finally, Dixie [49] distributes SQL queries among many database servers with replicated and partitioned tables, but requires manual replication and partitioning.

Distributed and concurrent data structures. Predating NOSQL systems, scalable distributed data structures [25] were proposed for constructing Internet services; these data structures provide an interface similar to key-value storage systems. Subsequently, distributed balanced trees [2, 42, 61], or DBTs, were proposed; however, previous DBTs, in contrast to YDBT, either do not provide ACID transactions [42] or suffer under contention [2, 61] (§6.1). Distributed skip graphs (e.g., [7]) are another data structure that can store indexes; however, they do not support transactions. Hyperspace hashing [16] provides simultaneous indexing of many attributes; however, Yesquel needs to index individual attributes, in which case hyperspace hashing reduces to partitioning by static ranges.

Concurrent B-trees are search data structures accessed by many threads in a multiprocessor system (e.g., [33, 36, 41, 43, 58, 59])—which is a different context from Yesquel's. YDBT draws ideas from this body of work. Back-down search (§4.1) is inspired by the give-up technique [59], which addresses the problem that a process may arrive at the wrong node during a search because of concurrent updates. The give-up technique detects this situation using fence intervals, and reacts by moving to some ancestor of the node. This is similar to our back-down search but differs in three ways: it starts every search from the root (we use client caches), it locks and unlocks nodes during the search (we do not lock), and it backtracks to an ancestor (we speculate and backtrack to a node in the cache). Delegated splits (§4.5) is related to splitting nodes at a later time [43, 59]; the former performs the splits of a node at a designated process, while the latter simply postpones the splits.

Other work. Liskov proposes the notion of algorithms that "depend on clocks for performance but not for correctness" [39]; this notion is later applied to optimistic concurrency control [1]. Yesquel's notion of clock safety (§4.6) is similar to that notion, but it is more precise and weaker, hence easier to satisfy. This is because (a) clock safety refers to safety not correctness (safety is a precise notion, correctness is vague), (b) clock safety refers to progress not performance (progress is weaker than performance), and (c) clock safety does not refer to clock dependency in any way (i.e., a protocol that does not depend on clocks for anything satisfies clock safety but not the notion in [39]).

9. Conclusion

Large-scale Web systems have been changing their storage back end from SQL to NOSQL. This transition destroys functionality (transactions, secondary indexes, joins, aggregation functions, subqueries), forcing developers to reimplement them for each Web application. While NOSQL systems have recently reincorporated some of the lost functionality, they still lag behind the richness of SQL. Yesquel's goal was to offer the power of SQL without sacrificing the performance and scalability of NOSQL. We believe that, at least in the domain of Web applications, Yesquel has met this goal.

Acknowledgements

We thank Dan Abadi, Miguel Castro, Curt Kolovson, Rama Kotla, Dennis Shasha, and our anonymous reviewers for many helpful comments that improved this paper. We are indebted especially to Roy Levin and Michael D. Schroeder for creating a rich and fertile research environment, where this work could blossom, at the late MSR Silicon Valley.

References

[1] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient optimistic concurrency control using loosely synchronized clocks. In *International Conference on Management of Data* (May 1995), pp. 23–34.

[2] AGUILERA, M. K., GOLAB, W., AND SHAH, M. A practical scalable distributed B-tree. *Proceedings of the VLDB Endowment 1*, 1 (Aug. 2008), 598–609.

[3] AGUILERA, M. K., LENERS, J. B., KOTLA, R., AND WALFISH, M. Yesquel: Scalable SQL storage for Web applications. In *International Conference on Distributed Computing and Networking* (Jan. 2015). Invited keynote presentation.

[4] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Distributed SQL query processing using key-value storage system, Dec. 2012. United States Patent Application 20140172898, filed 13 December 2012.

[5] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems 27*, 3 (Nov. 2009), 5:1–5:48.

[6] ALSBERG, P. A., AND DAY, J. D. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering* (Oct. 1976), pp. 562–570.

[7] ASPNES, J., AND SHAH, G. Skip graphs. *ACM Transactions on Algorithms 3*, 4 (Nov. 2007), 37.

[8] BERENSON, H., ET AL. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data* (May 1995), pp. 1–10.

[9] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] CHARRON-BOST, B., PEDONE, F., AND SCHIPER, A., Eds. *Replication: Theory and Practice*. Springer, 2010.

[11] CORBETT, J. C., ET AL. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation* (Oct. 2012), pp. 251–264.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation* (Dec. 2004), pp. 137–150.

[13] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's memory-optimized OLTP engine. In *International Conference on Management of Data* (June 2013), pp. 1243–1254.

[14] https://www.mapr.com/products/apache-drill.

[15] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems* (Sept. 2013), pp. 173–184.

[16] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A distributed, searchable key-value store for cloud computing. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Aug. 2012), pp. 25–36.

[17] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

[18] FLORATOU, A., MINHAS, U. F., AND ÖZCAN, F. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment 7*, 12 (Aug. 2014), 1295–1306.

[19] http://foundationdb.com.

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

[21] GOEL, A. K., POUND, J., AUCH, N., BUMBULIS, P., MACLEAN, S., FÄRBER, F., GROPENGIESSER, F., MATHIS, C., BODNER, T., AND LEHNER, W. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment 8*, 12 (Aug. 2015), 1716–1727.

[22] GRAEFE, G. Write-optimized B-trees. In *International Conference on Very Large Data Bases* (Aug. 2004), pp. 672–683.

[23] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *International Conference on Management of Data* (June 1996), pp. 173–182.

[24] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[25] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for Internet service construction. In *Symposium on Operating Systems Design and Implementation* (Oct. 2000), pp. 319–332.

[26] GUPTA, A., ET AL. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment 7*, 12 (Aug. 2014), 1259–1270.

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

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

[29] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2 (Feb. 2007), 141–259.

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

[31] KATE, B., KOHLER, E., KESTER, M. S., NARULA, N., MAO, Y., AND MORRIS, R. Easy freshness with Pequod cache joins. In *Symposium on Networked Systems Design and Implementation* (Apr. 2014), pp. 415–428.

[32] KORNACKER, M., ET AL. Impala: A modern, open-source SQL engine for Hadoop. In *Conference on Innovative Data Systems Research* (Jan. 2015).

[33] KUNG, H. T., AND LEHMAN, P. L. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (Sept. 1980), 354–382.

[34] http://en.wikipedia.org/wiki/LAMP_(software_bundle).

[35] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[36] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (Dec. 1981), 650–670.

[37] LEVANDOSKI, J. J., LOMET, D., MOKBEL, M. F., AND ZHAO, K. K. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research* (Jan. 2011), pp. 123–133.

[38] LEVIN, K. D., AND MORGAN, H. L. Optimizing distributed data bases: a framework for research. In *National computer conference* (May 1975), pp. 473–478.

[39] LISKOV, B. Practical uses of synchronized clocks in distributed systems. *Distributed Computing 6*, 4 (July 1993), 211–219.

[40] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the design and scalability of distributed shared-data databases. In *International Conference on Management of Data* (May 2015), pp. 663–676.

[41] LOMET, D. B., SENGUPTA, S., AND LEVANDOSKI, J. J. The Bw-tree: A B-tree for new hardware platforms. In *International Conference on Data Engineering* (Apr. 2013), pp. 302–313.

[42] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation* (Dec. 2004), pp. 105–120.

[43] MANOLOPOULOS, Y. B-trees with lazy parent split. *Information Sciences 79*, 1-2 (July 1994), 73–88.

[44] http://www.mediawiki.org.

[45] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment 3*, 1–2 (Sept. 2010), 330–339.

[46] http://memcached.org.

[47] MOHAN, C. Big data: Hype and reality. http://bit.ly/CMnMDS.

[48] http://www.mysql.com.

[49] NARULA, N., AND MORRIS, R. Executing Web application queries on a partitioned database. In *USENIX Conference on Web Application Development* (June 2012), pp. 63–74.

[50] NIELSEN, J. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[51] PAVLO, A., CURINO, C., AND ZDONIK, S. B. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *International Conference on Management of Data* (May 2012), pp. 61–72.

[52] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *Symposium on Operating Systems Design and Implementation* (Oct. 2010), pp. 279–292.

[53] http://prestodb.io.

[54] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., , AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *Symposium on Networked Systems Design and Implementation* (Apr. 2014), pp. 275–288.

[55] RAE, I., ROLLINS, E., SHUTE, J., SODHI, S., AND VINGRALEK, R. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment 6*, 11 (Aug. 2013), 1045–1056.

[56] REED, D. P. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems 1*, 1 (Feb. 1983), 3–23.

[57] http://www.scalearc.com.

[58] SEWALL, J., CHHUGANI, J., KIM, C., SATISH, N., AND DUBEY, P. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment 4*, 11 (Aug. 2011), 795–806.

[59] SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Transactions on Database Systems 13*, 1 (Mar. 1988), 53–90.

[60] SHUTE, J., ET AL. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment 6*, 11 (Aug. 2013), 1068–1079.

[61] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *Proceedings of the VLDB Endowment 5*, 9 (May 2012), 884–895.

[62] http://www.sqlite.org.

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

[64] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era (it's time for a complete rewrite). In *International Conference on Very Large Data Bases* (Sept. 2007), pp. 1150–1160.

[65] http://tajo.incubator.apache.org.

[66] TERRY, D., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AND AGUILERA, M. K. Transactions with consistency choices on geo-replicated cloud storage. Tech. Rep. MSR-TR-2013-82, Microsoft Research, Sept. 2013.

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

[68] TOMIC, A., SCIASCIA, D., AND PEDONE, F. MoSQL: An elastic storage engine for MySQL. In *Symposium On Applied Computing* (Mar. 2013), pp. 455–462.

[69] http://www.wikipedia.org.

[70] XIN, R. S., ROSEN, J., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: SQL and rich analytics at scale. In *International Conference on Management of Data* (June 2013), pp. 13–24.