Scalable Persistent Storage for Erlang: Theory and Practice

Amir Ghaffari
Natalia Chechina Phil Trinder
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, EH14 4AS, UK
http://www.release-project.eu

Jon Meredith
Basho Technologies, Inc.
http://basho.com
jmeredith@basho.com

Abstract

The many core revolution makes scalability a key property. The RELEASE project aims to improve the scalability of Erlang on emergent commodity architectures with 10^5 cores. Such architectures require scalable and available persistent storage on up to 100 hosts. We enumerate the requirements for scalable and available persistent storage, and evaluate four popular Erlang DBMSs against these requirements. This analysis shows that Mnesia and CouchDB are not suitable persistent storage at our target scale, but Dynamo-like NoSQL DataBase Management Systems (DBMSs) such as Cassandra and Riak potentially are.

We investigate the current scalability limits of the Riak 1.1.1 NoSQL DBMS in practice on a 100-node cluster. We establish for the first time scientifically the scalability limit of Riak as 60 nodes on the Kalkyl cluster, thereby confirming developer folklore. We show that resources like memory, disk, and network do not limit the scalability of Riak. By instrumenting Erlang/OTP and Riak libraries we identify a specific Riak functionality that limits scalability. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks. We conclude that Dynamo-style NoSQL DBMSs provide scalable and available persistent storage for Erlang in general, and for our RELEASE target architecture in particular.

Categories and Subject Descriptors [*Distributed Systems*]: Distributed databases; [*Performance of Systems*]: Fault tolerance

General Terms Experimentation, Measurement, Performance, Reliability

Keywords Scalability, Erlang, NoSQL Distributed Databases, Fault Tolerance, Eventual Consistency, Riak, Cassandra, Mnesia, CouchDB.

1. Introduction

As the number of cores in commodity hardware grows exponentially scalability becomes increasingly desirable as it gives systems the ability to available resources [5]. It is cost effective to construct large systems from commodity, i.e. low-cost but unreliable components [24]. In such an environment actor-based frameworks, and Erlang in particular, are increasingly popular for developing reliable scalable systems [2, 15].

The RELEASE project [6] aims to improve the scalability of Erlang on emergent commodity architectures with 10^5 cores. We anticipate that a typical architecture might comprise 5 clusters (virtual or physical), with approximately 100 hosts per cluster, and 32-64 cores per host. The project aims to improve the scalability of Erlang at the virtual machine, language, and infrastructure levels, and to supply profiling and refactoring tools. At the language level we seek to scale three aspects of Erlang:

1. *A scalable Computation model*, by providing alternatives to global data structures like the global name space and all-to-all connections between nodes.
2. *Scalable In-memory data structures*, and specifically more scalable Erlang Term Storage (ETS) tables.
3. *Scalable Persistent data structures*, i.e. scalable, and hence distributed DataBase Management Systems (DBMSs).

This paper investigates the provision of persistent data structures by studying the ability of Erlang distributed DBMS to scale on our target 10^5 core architecture. That is we investigate the scalability of Mnesia [14], CouchDB [23], and Dynamo-like NoSQL DBMSs like Riak [4], and Cassandra [17].

We start by enumerating the principles for scalable and available persistent data storage (Section 2). Crucially, availability is provided by replicating data on n independent hosts. Hence the hosts are the key architectural level, and the key issue for RELEASE is whether there are Erlang DBMSs that scale only approximately 100 nodes. We assess four DBMSs against the scalable persistent storage principles, and evaluate their suitability for large-scale architectures (Section 3). To evidence the scalability we benchmark the Riak NoSQL DBMS in practice (Section 4). Finally, we discuss findings and the future work (Section 5).

The paper makes the following research contributions.

- We present a theoretical analysis of common Erlang persistent storage technologies against the requirements for scalable and available persistent storage considering Mnesia, CouchDB, Riak, and Cassandra. Unsurprisingly, we find that Mnesia and CouchDB are not suitable persistent storage at our target scale, but Dynamo-like NoSQL DBMS such as Cassandra and Riak are (Section 3).
- We investigate the current scalability limits of the Riak 1.1.1 NoSQL DBMS using Basho Bench on 100-node cluster with

Technical Report HW-MACS-TR-0098.

800 cores. We establish for the first time scientifically the scalability limit of Riak 1.1.1 as 60 nodes on the Kalkyl cluster, thereby confirming developer folklore (Section 4.3).

- We show that resources like memory, disk and network do not limit the scalability of Riak (Section 4.5). By instrumenting the `global` and `gen_server` Erlang/OTP libraries we identify a specific Riak remote procedure call that fails to scale. The Basho developers had independently identified this bottleneck, and a second bottleneck. We outline how Riak 1.3 and 1.4 are refactored to eliminate the scalability bottlenecks (Section 4.5).

2. Scalable Persistent Storage

Distributed architectures such as clusters, Clouds, and Grids have emerged to provide high performance computing resources from low-cost unreliable components, i.e. commodity nodes and unreliable networks [24]. To better utilise these unreliable resources distributed database systems have been developed, i.e. data is distributed among a collection of nodes, and each node maintains a subset of the data. In this section we outline the principles of a highly scalable and available persistent storage, i.e. data fragmentation, data replication, partition tolerance, and query execution strategies [25].

Data fragmentation improves performance by spreading loads across multiple nodes and increases the level of concurrency (Fig. 1). A scalable fragmentation approach should have the following features:

1. Decentralized model. In a decentralised model data is fragmented between nodes without a central coordination. The models show a better throughput in large systems, and have a higher availability in comparison with the centralised models due to eliminating a single point of failures [13].
2. Load balancing, i.e. even distribution of data between the nodes. A desirable load balancing mechanism should take the burden of load balancing off the developer shoulders. For example, such decentralized techniques as consistent hashing [20] may be employed.
3. Location transparency. Placing fragments on a proper location in a large scale system is very difficult to manage. Therefore, the placement should be carried out automatically and systematically without a developer interference. Moreover, a reorganisation of database nodes should not impact the programs that access the database [11].
4. Scalability. A node departure and arrival should only affect the node immediate neighbours, and the remaining nodes should be unaffected. A new node should receive approximately the same amount of data as other available nodes have, and when a node goes down, its load should be evenly distributed between the remaining available nodes [13].

Data Replication improves performance of read-only queries by providing data from the nearest replica (Fig. 2). The replication may also increase the system availability by removing a single

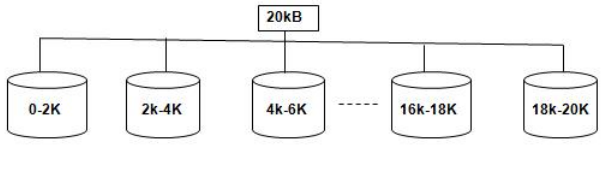

Figure 1: An Even Fragmentation of 20KB Data over Ten Nodes

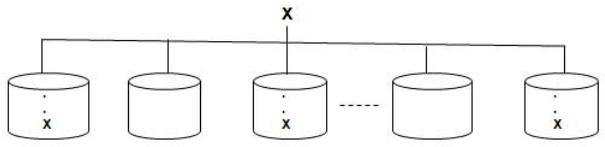

Figure 2: Replication of Record X on Three Nodes

point of failures [25]. A scalable mechanism for replication should have the following features:

1. Decentralized model. Data should be replicated to nodes without using a concept of a master, so each node has a full DBMS functionality [13]. A P2P model is desirable because each node is able to coordinate the replication, and improve overall performance by distributing computational and network traffic loads over the available nodes.
2. Location transparency. The placement of replicas should be handled systematically and automatically [11]. However, managing the location of replicas can be a difficult task in large-scale systems.
3. Asynchronous replication, i.e. `write` command is considered complete as soon as the local storage acknowledges it without necessity to wait for the remote acknowledgment [18]. In large-scale systems with slow communication, such as wide area networks, waiting for the confirmation can be time consuming. Many Internet-scale applications, e.g. email and social network services, may be able to tolerate some inconsistency among the replicas to provide a better performance. The approach may violate the consistency of data over time, so *eventual consistency* can be used to address the problem. Eventual consistency [27] is a specific form of weak consistency where updates are propagated throughout the system, and eventually all participants have the last updated value. Domain Name System (DNS) is the most popular system that employs eventual consistency.

Partition Tolerance is essential to cope with node failures and network partitions in loosely coupled large-scale distributed systems. A highly available system should continue to operate despite a loss of connections between some nodes. The CAP theorem [18] states a database cannot simultaneously guarantee consistency, availability, and partition tolerance (Fig. 3). Thus, to achieve partition tolerance and availability strong consistency must be sacrificed. The eventual-consistency mechanism can improve availability by providing weakened consistency, i.e. in case some nodes fail or become unreachable while an update is being executed the changes will be applied to the failed nodes as soon as they recover from the failure.

Query Execution Strategies provide mechanisms to retrieve a specific information from a database. In addition to a good performance a desirable query processing approach hides low-level details of replicas and fragments [25]. A scalable query strategy should have the following features:

1. Local execution. On geographically distributed large-scale systems where data is fragmented over nodes query response time may become very high due to communication. Scalable techniques like MapReduce [12] reduce the amount of transferring data between the participating nodes in a query execution by *local processing*, i.e. passing the queries to where the data lives rather than transferring data to a client. After the local processing the results of local executions are combined into a single output.

2. Parallelism. In addition to improving the performance the local processing increases parallel execution of large computations by dividing the query into sub-queries and spreading them among multiple nodes. Thus, a good strategy should exploit the local resources as much as possible to achieve the maximum efficiency.
3. Fault tolerance. Since large-scale databases are distributed on large clusters of nodes a node failure is very common. Therefore, a query strategy should tolerate communication and node failures. In case of a failure the query coordinator should mask the failure by running the query on the available replicas.

3. Scalability of Erlang DBMSs

Recently, there has been a lot of interest in NoSQL databases to store and process large-scale data sets, such as Amazon Dynamo [13], Google BigTable [7], Facebook Cassandra [22], Yahoo! PNUTS [9]. These companies have rejected traditional relational DBMSs because these systems could not provide high scalability, high availability, and low latency for large scale unreliable distributed environments.

In this section we analyse the following four popular NoSQL DBMSs for Erlang systems: Mnesia (Section 3.1), CouchDB (Section 3.2), Riak (Section 3.3), and Cassandra (Section 3.4). The databases are evaluated in terms of scalability and availability against the principles outlined in Section 2.

3.1 Mnesia

Mnesia is a distributed DBMS written in Erlang for industrial telecommunications applications [14]. The Mnesia data model consists of tables of records. Attributes of each record can store arbitrary Erlang terms. Mnesia provides ACID (Atomicity, Consistency, Isolation, Durability) transactions, i.e. either all operations in a transaction are applied on all nodes successfully, or, in case of a failure, no node effected. In addition, Mnesia guarantees that transactions which manipulate the same data records do not interfere with each other. To read from, and write to, a table through a transaction Mnesia sets and releases locks automatically. Fault tolerance is provided in Mnesia by replicating tables on different Erlang nodes. In Mnesia replicas are placed explicitly, e.g. Code 1 shows a `student` table replicated on three Erlang VMs.

Code 1: An Explicit Placement of Replicas in Mnesia

```
mnesia:create_table(student, [{disc_copies,
 [node1@sample_domain, node2@sample_domain,
node3@sample_domain]},{type, set},{attributes,
[id,fname,lname,age]},{index,[fname]}]).
```

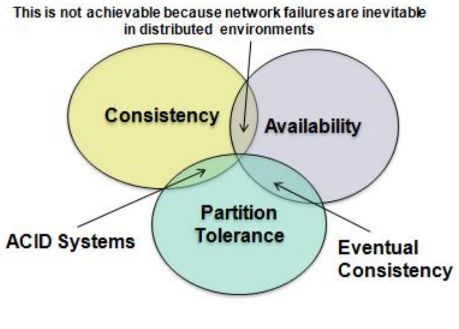
This is not achievable because network failures are inevitable in distributed environments
Consistency
Availability
Partition Tolerance
ACID Systems
Eventual Consistency

Figure 3: CAP Theorem [18]

In general, to read a record only one replica of that record is locked (usually the local one), but to update a record all replicas of that record are locked and must be updated. This can become a bottleneck for write operations when one of the replica is not reachable due to node or network failures. To address the problem Mnesia offers *dirty operations* that manipulate tables without locking all replicas. However, dirty operations do not provide a mechanism to eventually complete the update on all replicas, and this may lead to data inconsistency.

Another limitation of Mnesia is the size of tables. The limitation is inherited from DETS tables, and since DETS tables use 32 bit file offsets, the largest possible Mnesia table per an Erlang VM is 2GB. To cope with large tables Mnesia introduces a concept of *table fragmentation*. A large table can be split into several smaller fragments on different Erlang nodes. Mnesia employs a hash function to compute the hash value of a record key, and then that value is used to determine the fragment the record belongs to. The downside of the fragmentation mechanism is that the placement of fragments should be specified explicitly by the user. Code 2 shows an explicit placement of fragments in Mnesia.

Code 2: An Explicit Placement of Fragments in Mnesia

```
mnesia:change_table_frag(SampleTable,
 {add_frag, List_of_Nodes}).
```

Query List Comprehensions (QLCs) is an Erlang query interface to Mnesia. QLCs are similar to ordinary list comprehensions in Erlang programming language, e.g. Code 3 returns `name` of student whose `id=1`.

Code 3: A Query Example in Mnesia

```
Query = query [S.lname ||
                S <- table(student), S.id == 1] end,
Result = mnesia:transaction(
                 fun() -> mnesyne:eval(Query)
          end).
```

A summary of Mnesia limitations for large-scale systems is as follows:

- Explicit placement of fargments and replicas.
- Limitation in size of tables.
- Lack of support for eventual consistency.

3.2 CouchDB

CouchDB (Cluster Of Unreliable Commodity Hardware) is a schema-free document-oriented database written in Erlang [23]. Data in CouchDB is organised in a form of a document. Schemaless means that each document can be made up of an arbitrary number of fields. A single CouchDB node employs a B-tree storage engine that allows to handle searches, insertions, and deletions in logarithmic time. Instead of traditional locking mechanisms for concurrent updates CouchDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent access to the database. With MVCC a system is able to run at full speed all the time, even when a large number of clients uses the system concurrently. In CouchDB view creation and report aggregation is implemented by joining documents using Map/Reduce technique.

Data fragmention over nodes is handled by Lounge – a proxy-based partitioning/clustering framework for CouchDB [1]. To find the shard where a document should be stored Lounge applies a consistent hash function to the document's ID. Lounge does not operate on all CouchDB nodes. In fact, Lounge is a web proxy that distributes HTTP requests to CouchDB nodes. Thus, to remove a single point of failures multiple instances of Lounge should be run, i.e a multi-server model.

CouchDB replication system synchronizes all copies of the same database by sending the most recent changes to all other replicas. Replication is an unidirectional process, i.e. the changed documents are copied from one replica to another and not automatically vice versa. Currently, replica placements are handled explicitly (Code 4).

Code 4: An Explicit Placement of Replicas in CoachDB

```
POST /_replicate HTTP/1.1
{"source":"http://localhost/database",
"target":"http://example.org/database",
 "continuous":true}
```

In Code 4, `''continuous'':true` means that CouchDB will not stop the replication and will automatically send new changes of the `source` to the `target` by listening to the CouchDB API changes. CouchDB has eventual consistency, i.e. document changes are periodically copied to replicas. A conflict occurs when a document has different information on different replicas. In case of a conflict between replicas CouchDB employs an automatic conflict detection mechanism. CouchDB does not attempt to merge the conflicting visions but only attempts to find the latest version of the document. The latest version is the winning version. The other versions are kept in the document's history, and client applications can use them to resolve the conflict in an alternative way.

In summary, CouchDB has the following limitations that can be a bottleneck for large-scale systems:

- Explicit placement of fragments and replicas.
- Multi-server model to coordinate fragmentation and replication.
- Lounge program that handles data partitioning is not a part of every CouchDB node.

3.3 Riak

Riak is a NoSQL, schemaless, open source, distributed key/value data store primarily written in Erlang [4]. Riak is scalable, available, and fault tolerant database suitable for large-scale distributed environments, such as Clouds and Grids. Riak is highly fault tolerant due to its masterless structure that offers no single point of failures. Riak is commercially proven and is used by large and well known companies and organizations such as Mozilla, Ask.com, AOL, DotCloud, GitHub.

In Riak data is stored using key/value pairs. A key/value pair is stored in a bucket. A values can be retrieved using its unique key. Data fragmentation in Riak is handled implicitly using a consistent hashing technique [4]. Consistent hash dynamically distributes data over nodes as nodes join and leave the system.

For availability Riak uses a replication and a hand-off technique. By default each data bucket in Riak is replicated on three different nodes. However, the number of replicas (N) is a tunable parameter and can be set for every bucket. Other tunable parameters are read quorum (R) and write quorum (W). A quorum is the number of replicas that must respond to a read or write request before it is considered successful. If $R + W > N$ then Riak provides a strong consistency which guarantees that a subsequent accesses returns the previously updated value. When $W + R <= N$ Riak provides a weak consistency and some nodes may keep outdated data.

When a communication with a node is lost temporarily due to a node failure or a network partitioning a hand-off technique is used, i.e. neighbouring nodes take over the duties of the failed node. When the failed node comes back a Merkle tree is used to determine the records that need to be updated [4]. Each node has its own Merkle tree for the keys it stores. Merkle trees reduce the amount of data needed to be transferred to check inconsistencies between replicas.

Riak provides eventual consistency, i.e. an update is propagated to all replicas asynchronously. However, under certain conditions such as node failures and network partitioning, updates may not reach all replicas. Riak employs a vector clock (or vclock) mechanism to handle the inconsistencies by reconciling the older version and the divergent versions.

The default Riak backend storage is Bitcask. Although Bitcask provides low latency, easy backup, restore, and is robust in the face of crashes but it has one notable limitation. Bitcast keeps all keys in RAM and therefore can store a limited number of keys per node. For this reason Riak users may use other storage engines to store billions of records per node.

LevelDB is a fast key-value storage library written at Google that has no Bitcask RAM limitation. LevelDB provides an ordered mapping from keys to values whereas Bitcask is a hash table. LevelDB supports atomic batch of updates that may also be used to speed up large updates by placing them into the same batch. A LevelDB database may only be opened by one process at a time. There is one file system directory per each LevelDB database where all database content is stored. To improve the performance adjacent keys are located in the same block. A block is a unit of data used to transfer data to and from the persistent storage. The default block size is approximately 8192 bytes. Each block is individually compressed before being written to persistent storage, however, compression can also be disable. It is possible to force a checksum verification of all data that is read from the file system. Eleveldb is an Erlang wrapper for LevelDB included in Riak, and does not require a separate installation. LevelDB read access can be slower in comparison with Bitcask because LevelDB tables are organized into a sequence of levels. Each level stores approximately ten times as much data as the level before it. For example if 10% of the database fits in memory, one search is required to reach the last level. But if 1% fits in memory, LevelDB will require two searches.

The theoretical analysis shows that Riak meets scalability requirements of a large-scale distributed system. Here we summarize the requirements:

- Implicit placement of fragments and replicas.
- Bitcask backend has a limitation in size of tables but LevelDB backend has no such limitation, and can be used instead.
- Eventual consistency that consequently brings a good level of availability.
- No single point of failures as peer to peer (P2P) model is used.
- Scalable query execution approach that supports MapReduce queries.

3.4 Cassandra

There are other distributed DBMSs which are not written in Erlang but Erlang applications can access them by using a client library. The Apache Cassandra is a highly scalable and available database written in Java recommended for commodity hardware or cloud infrastructure [17]. Cassandra is used at Twitter, Cisco, OpenX, Digg, CloudKick, and other companies that have large data sets. Cassandra offers an automatic, master-less and asynchronous mechanism for replication. Cassandra has a decentralized structure where all nodes in a cluster are identical, and therefore, there is no single point of failures and no network bottlenecks. Cassandra provides a ColumnFamily-based data model which is richer than typical key/-value systems. Large scale queries can be run on a Cassandra cluster by using Hadoop MapReduce. Hadoop runs MapReduce jobs to retrieve data from Cassandra by installing a TaskTracker on each Cassandra node. This is an efficient way to retrieve data because each TaskTracker only receives queries for data that the local node is the primary replica. This avoids an overhead of the Gossip protocol.

Since both Riak and Cassandra are inspired by Amazon's description of Dynamo [13], their methods for load-balancing, replication, and fragmentation are similar. So we do not repeat details here. Erlang applications employ the Thrift API to use Cassandra. There are also some client libraries for common programming languages such as Erlang, Python, Java, recommended to use instead of raw Thrift. Cassandra meets the general principles of scalable persistent storages, i.e.

- Implicit placement of fragments and replicas.
- ColumnFamily-based data model.
- Eventual consistency.
- No single point of failures due to using a P2P model.
- Scalable query execution approach by integrating Hadoop MapReduce.

3.5 Discussion

The theoretical evaluation shows that Mnesia and CouchDB have some scalability limitations, i.e. implicit placement of replicas and fragments, single point of failures due to lack of P2P model (Sections 3.1 and 3.2). Dynamo-style NoSQL DBMS like Riak and Cassandra do have a potential to provide scalable storage for large distributed architecture as required by the RELEASE project (Sections 3.3 and 3.4). In Section 4 we investigate Riak scalability and availability in practice.

4. Riak Scalability and Availability

This section investigates *scalability* of Riak DBMS, i.e. how system throughput increases by adding Riak nodes. In addition to scalability we measure Riak *availability* and *elasticity*. In an availability benchmark we examine the effect of node failures. Elasticity is an ability to cope with loads dynamically when the number of nodes in the cluster changes.

We use the popular Basho Bench benchmarking tool for NoSQL DBMS [3]. Basho Bench is an Erlang application that has a pluggable driver interface and can be extended to serve as a benchmarking tool for data stores.

4.1 Experiment Setup

Platform. The benchmarks are conducted on the Kalkyl cluster [26]. The Kalkyl cluster consists of 348 nodes with 2784 64-bit processor cores connected via 4:1 oversubscribed DDR Infiniband fabric. Each node comprises Intel quad-core Xeon 5520 2.26 GHz processors with 8MB cache, and has 24GB RAM memory and 250 GB hard disk. The Kalkyl cluster runs Scientific Linux 6.0, a Red Hat Enterprise Linux. Riak data is stored on the local hard drive of each node.

Parameters. In the experiments we use Riak version 1.1.1. The number of partitions, sometimes referred to as virtual nodes or vnodes, is 2048. In general, each Riak node hosts N_1 number of vnodes, i.e.

$$N_1 = \frac{N_{vnodes\ in\ the\ cluster}}{N_{nodes\ in\ the\ cluster}}$$

Riak documentation recommends 16-64 vnodes per node, e.g. 64-256 vnodes for a 4-node cluster. In the experiments we keep the default setting for replication, i.e. data is replicated to three nodes on the cluster. The number of replicas that must respond to a read or write request is two, which is also the default value.

4.2 How Does the Benchmark Work?

We have conducted the experiments on a cluster where each node can be either a *traffic generator* or a *Riak node*. A traffic generator runs one copy of Basho Bench that generates and sends commands to Riak nodes. A Riak node contains a complete and independent copy of the Riak package which is identified by an IP address and a port number. Fig. 4 shows how traffic generators and Riak nodes are organized inside a cluster. There is one traffic generator for every three Riak nodes.

Each Basho Bench application creates and runs in parallel 90 Erlang process, i.e. workers (Fig. 5). We picked this particular number of processes because it seems large enough to keep traffic generators busy. Then, every worker process randomly selects an IP address from a predefined list of Riak nodes. A worker process randomly selects one of the following database operation: `get`, `insert`, or `update`, and submits the corresponding HTTP or Protocol Buffer command to the selected IP address and port number. The default port number for the HTTP communication is `[8098]`,

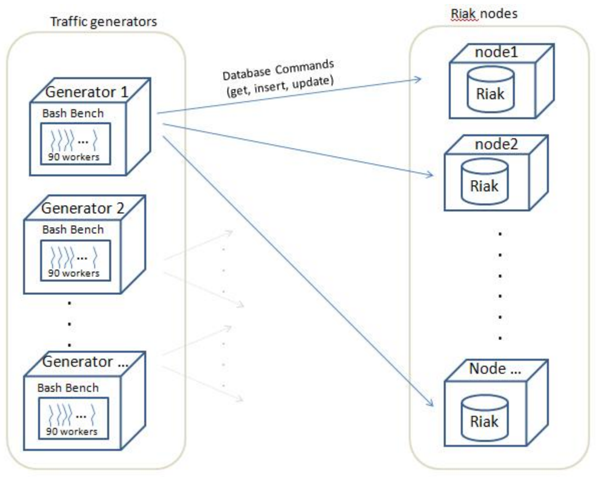

Figure 4: Riak Nodes and Traffic Generators in Basho Bench

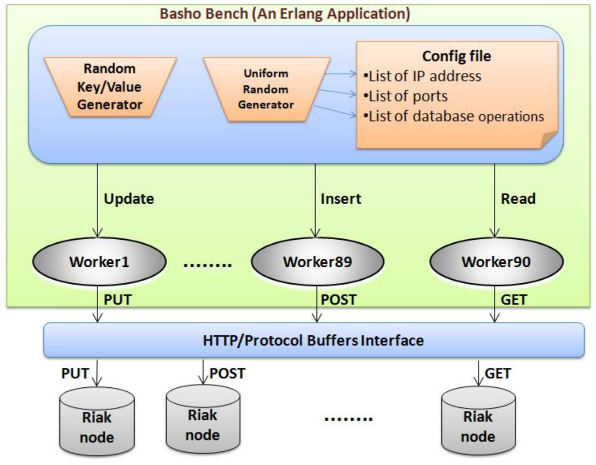

Figure 5: Structure of Basho Bench

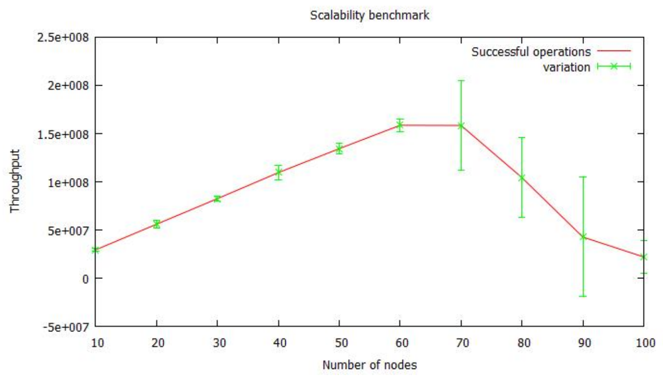

Figure 6: Riak Throughput vs. Number of Nodes

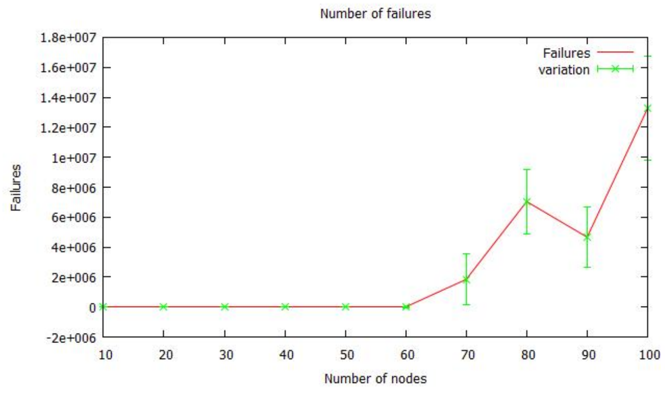

Figure 7: Number of Failures vs. Number of Nodes

and for the protocol buffer the number is [8087]. A list of database operations and corresponding HTTP commands is as follows:

1. `Get` corresponds to the HTTP GET command
2. `Insert` corresponds to the HTTP POST command
3. `Update` corresponds to the HTTP PUT command

4.3 Scalability Measurements

We measure how throughput rises as the number of Riak nodes increases, i.e. on 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100-node clusters. In the experiments the traffic generators issue as much database commands as the cluster can serve. We repeat every experiment three times. Fig. 6 depicts the mean values. The results show that Riak scales linearly up to 60 nodes, but it does not scale beyond 60 nodes. In Fig. 6 the green line represents variation, and it is very small for the clusters with up to 60 nodes, but significantly increases when the number of nodes increases.

Fig. 7 shows the number of failed operations. In a cluster with up to 60 nodes the number of failures is 0, but when we increase the size of the cluster failures emerge. For example there are 2 million failures on a 70-node cluster, i.e. 1.3% of failures to the total number of operations. From the log files the reason of the failures is timeout. Fig. 8 shows the mean latency of successful operations. The mean latency of operations in a 60-node cluster is approximately 21msec, which is much less than timeout, i.e. 60sec.

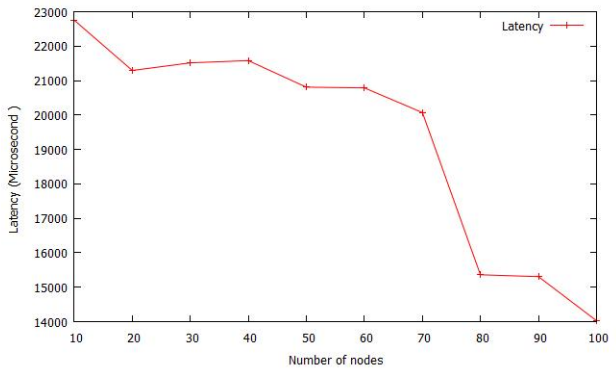

Figure 8: Mean Command Latency (Sec. 4.2) vs. Number of Nodes

4.4 Resource Scalability Bottlenecks

To identify possible bottlenecks of the Riak scalability we measure the usage of random access memory (RAM), disk, cores, and network.

Fig. 9 shows RAM utilisation on Riak nodes and traffic generators depending on the total number of nodes. The maximum memory usage is 720MB out of 24GB of total RAM memory, i.e. less than 3%. We conclude that RAM is not a bottleneck for Riak scalability.

Fig. 10 shows percentage of time that a disk spends serving requests. The maximum usage of disk is approximately 10% on a 10-node cluster. Disk usage for traffic generators is approximately 0.5%. We conclude that disk is not a bottleneck for Riak scalability either.

Fig. 11 shows mean core utilisation on 8-core Riak nodes and traffic generators. As Riak is P2P we anticipate that all nodes have similar mean utilisation. The maximum mean utlisation is approximately 550%, so 5.5 cores of the 8 cores are used. We have not yet measured the maximum load on all cores to be sure that no core is overloaded, but the Figure shows that there are cores available.

When profiling the network we count the number of the following packets: *sent*, *received*, and *retransmitted*. Fig. 12 and 13 show results for traffic generators and Riak nodes respectively. An increase of the number of sent and received packets is consistent with the increase of throughput. The number of sent and received packets increases linearly up to 60 nodes and beyond that there is a significant decrease. To check whether there is a TCP incast [16, 21]

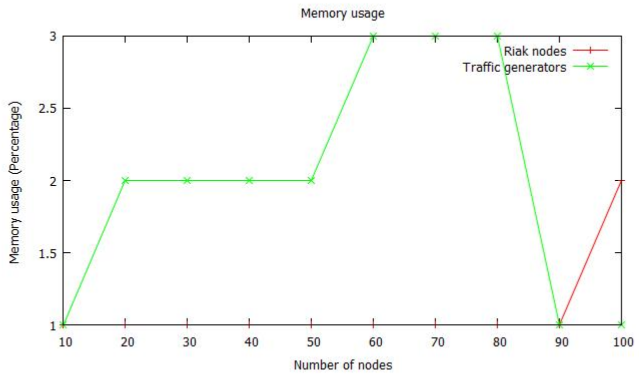

Figure 9: Maximum RAM Usage vs. Number of Nodes

Module	Function		Runtime on a cluster of			
			4 nodes	8 nodes	16 nodes	24 nodes
rpc	call	T_{mean}	1953	2228	8547	15180
		N_{mean}	106591	77468	49088	31104
riak_kv_put_fsm_sup	start_put_fsm	T_{mean}	2019	2284	8342	13461
		N_{mean}	104973	82070	48689	33898

Table 1: Two the Most Time-consuming Riak Functions

we count the number of retransmitted packets. In general, TCP incast occurs when a number of storage servers send a huge amount of data to a client, and Ethernet switch is not able to buffer the packets. When TCP incast strikes the number of lost packets increases and consequently causes a growth in the number of *retransmitted* packets. However, counting of the retransmitted packets shows that TCP incast has not occurred during the benchmark. The maximum number of retransmitted packets is 200 packets which is negligible.

A comparison of network traffic between traffic generators and Riak nodes shows that Riak nodes produce five times more network traffic than traffic generators. For example on a 10-node cluster traffic generators send 100 million packets, whereas Riak nodes send 500 million packets on the cluster of the same size. The reason is due to the fact that to replicate and maintain data Riak nodes in addition to communicating with generators also communicate between each other.

4.5 Riak Software Scalability

The profiling above reveals that Riak scalability is not bound by resources like memory or disk, so we need to investigate the scalability of Riak software. It is well known that the frequency of global operations limits scalability. Indeed in separate work we show the limitations imposed by the percentage of global operations in Distributed Erlang [19].

While it is not feasible to search the entire Riak codebase for global operations in the form of iterated P2P operations, we investigated two likely sources of global operations.

1. We instrument the global name registration module `global.erl` to identify the number of calls, and the time consumed by each global operation. The result shows that Riak makes no `global.erl` calls.
2. We also instrument the genserver module `gen_server.erl`. Of the 15 most time-consuming operations, only the time

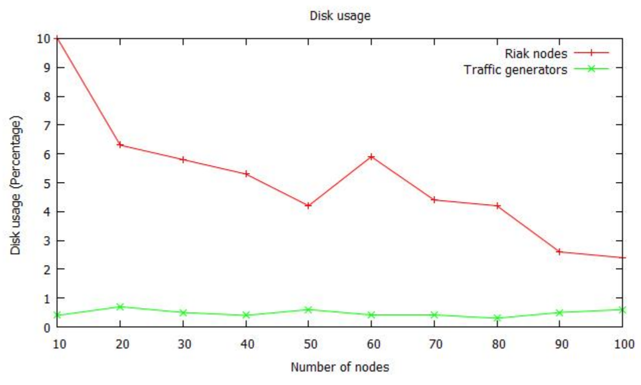

Figure 10: Maximum Disk Usage vs. Number of Nodes

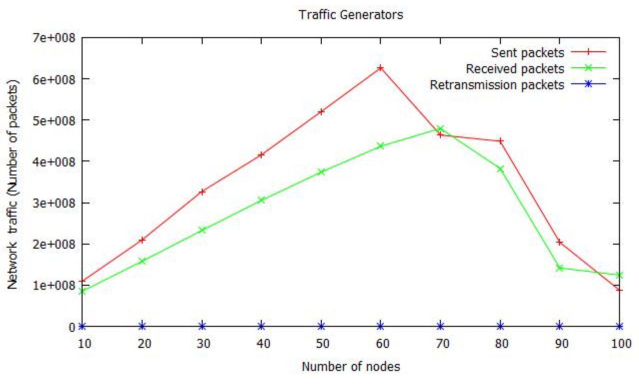

Figure 12: Network Traffic of Traffic Generators

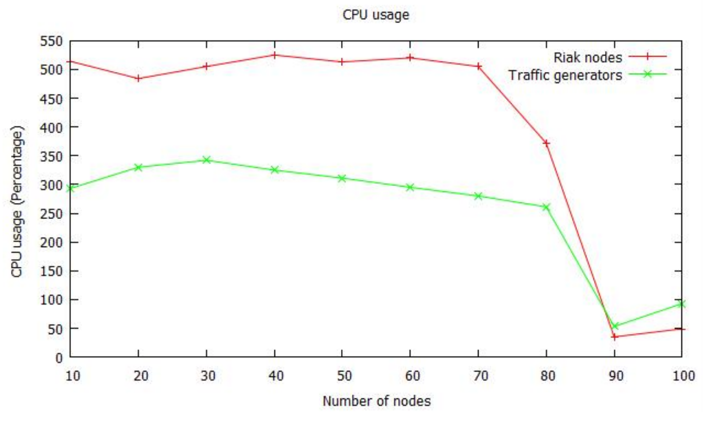

Figure 11: Core Usage on 8-core Nodes

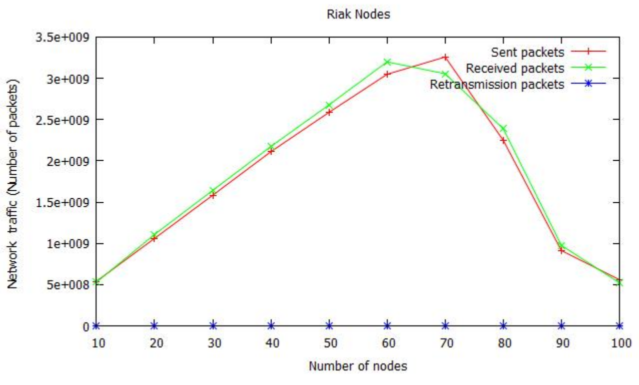

Figure 13: Network Traffic of Riak Nodes

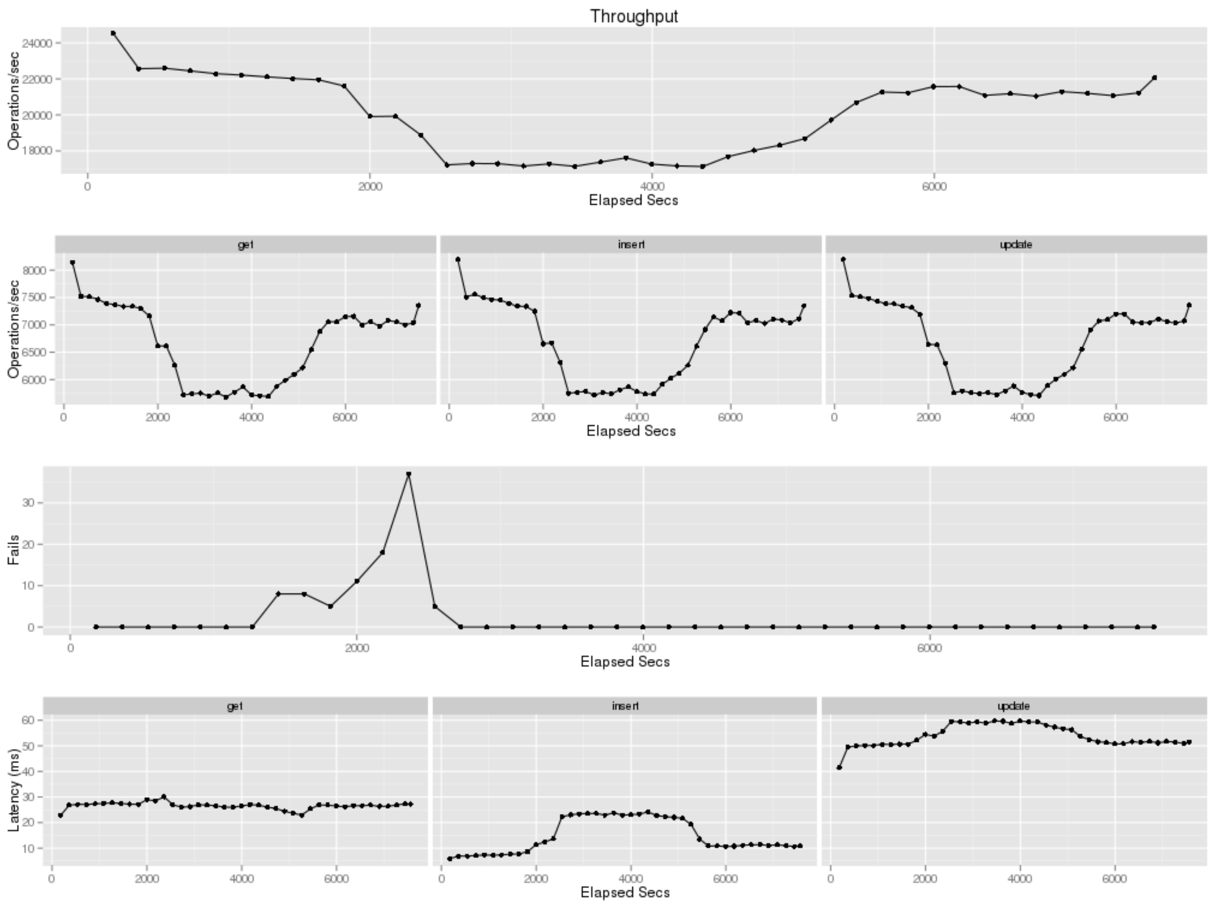

Figure 14: Throughput and Latency in the Availability and Elasticity Benchmark

of `rpc:call` grows with cluster size. Moreover, of the five Riak RPC calls, only `start_put_fsm` function from module `riak_kv_put_fsm_sup` grows with cluster size (Table 1). T_{mean} shows the mean time that each function call takes to be completed in microseconds and N_{mean} is the mean number of times that a function is called during 5 minutes benchmarking in 3 executions.

Independently, Basho [3] have analysed Riak scalability and had identified the `riak_kv_get/put_fsm_sup` issue, together with a scalability issue with statistics reporting. To improve the Riak scalability Basho applied a number of techniques and introduced new library `sidejob` (https://github.com/basho/sidejob). These modifications are available in Riak version 1.3 and upcoming version 1.4. An overview of the modifications is presented below.

In Riak version 1.0.x through 1.2.x creating get/put FSM (Finite State Machine) processes go through two supervisor processes, i.e. `riak_kv_get/put_fsm_sup`. The supervisors are implemented as single-process `gen_servers` and become a bottleneck under heavy load, exhibiting build up in message queue length. In Riak version 1.3 get/put FSM processes are created directly on the external API-handling processes that issue the requests, i.e. `riak_kv_pb_object` (protocol buffers interface) or `riak_kv_wm_object` (REST interface). To track statistics and unexpected process exits without supervisors the get/put FSM processes registere themselves asynchronously with a new monitoring process `riak_kv_getput_mon`. Similarly, to avoid another single process bottleneck, Basho replaced RPC calls in the put FSM implementation that forwards data to responsible nodes with direct `proc_lib:spawn`. This avoids another single-process bottleneck through the `rex` server used by `rpc:call`. Riak version 1.3 also has some refactoring to clean up unnecessary old code paths and do less work, e.g. the original map/reduce cache mechanism that was replaced by `riak_pipe` was still having cache entries ejected on every update.

In Riak version 1.4 the get/put FSM spawn mechanism was replaced by a new mechanism presented in the sidejob library. The library introduces a parallel mechanism for spawning processes and enforces an upper limit on the number of active get/put FSMs to avoid process table exhaustion when the node is heavily loaded. Exhaustion of the process table has caused a cascade cluster failure in a production deployments. The library was also used to eradicate the bottleneck caused by the statistics reporting process.

4.6 Availability and Elasticity

Distributed database systems must maintain availability despite network and node failures. Another important feature of distributed systems is elasticity. Elasticity means an equal and dynamic distribution of the load between nodes when resources (i.e. Riak nodes) either removed or added to the system [10]. To benchmark Riak availability and elasticity we run seven generators and twenty Riak nodes. During the benchmark the number of generators remains constant (seven) but the number of riak nodes changes. Fig. 15

shows that during the first 30 minutes of the benchmark there are 20 Riak nodes. We choose 30 minutes because we want to be sure that the system is in a stable state. After 30 minutes nodes go down every two minutes. In total 9 nodes go down until minute 48, i.e. approximately50% of failures. Between minutes 48 and 78 the system has 11 Riak nodes. After minute 78 nodes come back every two minutes. Thus, after minute 96 all 20 nodes are back. After minute 96 the benchmark runs on 20 Riak nodes for another 30 minutes.

Fig. 14 shows that when the cluster loses 50% of its Riak nodes (between minutes 30 and 48) Riak throughput decreases and the number of failures grows. However, in the worst case the number of failures is 37 whereas the number of successful operations is 3.41 million. Between minutes 48 and 78 the throughput does not change dramatically, and during and after adding new nodes the throughput grows. Thus, we conclude that Riak has a very good level of availability and elasticity.

4.7 Summary

Riak version 1.1.1 scales up to approximately 60 nodes linearly on the Kalkyl cluster (Section 4.3). But beyond 60 nodes throughput does not scale and timeout errors emerge (Fig. 6 and 7). To identify the Riak scalability problem we profiled RAM, disk, cores, and network (Section 4.4). The results of RAM and disk profiling show that these cannot be a bottleneck for Riak scalability, maximum RAM usage is 3%, and the maximum disc usage is 10%. Maximum core usage on 8 core nodes is 5.5 cores, so cores are available. The network profiling shows that the number of retransmitted packets is negligible in comparison with the total number of successfully transmitted packets, i.e. 200 packets out of $5 \cdot 10^8$ packets.

Our observations of Riak, together with correspondance with the Basho developers reveal that the scalability limits are due to single supervisor processes such as `riak_kv_get/put_fsm_sup` processes and `rex` process in `rpc.erl` (Section 4.5). These scalability obstacles were eliminated in Riak versions 1.3 and 1.4. Riak shows very good availability and elasticity (Section 4.6). After losing 9 Riak nodes only 37 failures occured, whereas the number of successful operations was 3.41 million. When failed nodes come back up the throughput grows.

5. Conclusion and Future Work

We have rehearsed the requirements for scalable and available persistent storage (Section 2), and evaluated four popular Erlang DBMS against these requirements. We conclude that Mnesia and CouchDB are not suitable persistent storage at our target scale, but Dynamo-style NoSQL DBMS like Cassandra and Riak have the potential to be (Section 3).

We have investigated the current scalability limits of the Riak version 1.1.1 NoSQL DBMS using Basho Bench on 100-node cluster with 800 cores. We establish for the first time scientifically the scalability limit of Riak 1.1.1 as 60 nodes on the Kalkyl cluster, thereby confirming developer folklore.

We show that resources like memory, disk, and network do not limit the scalability of Riak (Section 4.5). By instrumenting the `global` and `gen_server` OTP libraries we identify a specific Riak remote procedure call that fails to scale. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks (Section 4).

We conclude that Dynamo-like NoSQL DBMSs provide scalable and available persistent storage for Erlang. The RELEASE target architecture requires scalability onto 100 hosts, and we are confident that Dynamo-like NoSQL DBMSs will provide it. Specifically the Cassandra interface is available and Riak 1.1.1 already provides scalable and available persistent storage on 60 nodes. Moreover the scalability of Riak is much improved in versions 1.3 and 1.4 and will continue to improve.

In ongoing work we are investigating the scalability limitations of Distributed Erlang, and developing techniques to improve those limitations [8]. The technologies we introduce may further improve the scalability of persistent storage engines implemented in Distributed Erlang.

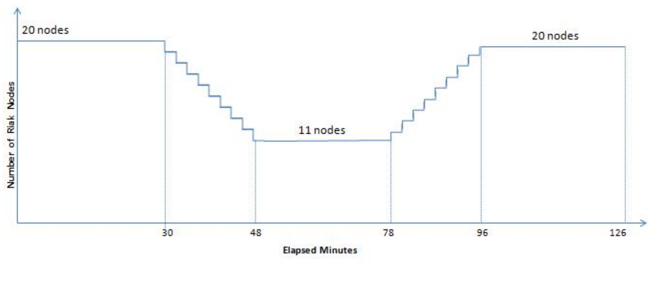

Figure 15: Availability and Elasticity Time-line

Acknowledgments

This work has been supported by the European Union grant IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software'. The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under RELEASE Project.

We thank Justin Sheehy from Basho Technologies for useful technical insights.

References

[1] Anderson, J. C., Lehnardt, J., Slater, N. (2010). CouchDB: The Definitive Guide. (M. Loukides, Ed.) (First Ed). OReilly Media, Inc.

[2] Armstrong, J. (2007). Programming Erlang: Software for a Concurrent World (1st Ed). Pragmatic Bookshelf.

[3] Basho Technologies, B. (2012). Benchmarking. Basho wiki. Retrieved from http://wiki.basho.com/Benchmarking.html

[4] Basho Wiki, B. (2012). Concepts. Basho Technologies, Inc. Retrieved from http://wiki.basho.com/Concepts.html

[5] Bondi, A. B. (2000). Characteristics of Scalability and Their Impact on Performance. Proceedings of the 2nd international workshop on Software and performance ACM New York, NY, USA, pages 195-203.

[6] Boudeville, O., Cesarini, F., Chechina, N., Lundin, K., Papaspyrou, N., Sagonas, K., Thompson, S., et al. (2012). RELEASE: A High-level Paradigm for Reliable Large-scale Server Software. In proceedings of the Symposium on Trends in Functional Programming - St Andrews University, UK.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., et al. (2008). Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) - New York, NY, USA, 26(2).

[8] Chechina N., Trinder P., Ghaffari A., Green R., Lundin K., and Virding R. The Design of Scalable Distributed Erlang. In draft proceedings of the Symposium on Implementation and Application of Functional Languages 2012 (IFL'12), 2012.

[9] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., et al. (2008). PNUTS: Yahoo!'s Hosted Data Serving Platform. Proceedings of the VLDB Endowment, 1(2), pages 1277-1288.

[10] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. SoCC'10 Proceedings of the 1st ACM symposium on Cloud computing New York, NY, USA, pages 143-154.

[11] Connolly, T., Begg, C. (2005). Database Systems: A Practical Approach to Design, Implementation, and Management (Forth Ed). Pearson Education Limited.

[12] Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - USENIX Association Berkeley, CA, USA.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., et al. (2007). Dynamo: Amazon's Highly Available Key-value Store. Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - ACM New York, NY, USA.

[14] Ericsson. (2012). Mnesia. Retrieved from http://www.erlang.org/doc/man/mnesia.html

[15] Ericsson. (2012). Who uses Erlang for product development? Retrieved from http://www.erlang.org/faq/introduction.html#id49610

[16] Fritchie, S. L. (2012). TCP incast: What is it? How can it affect Erlang applications? Retrieved from http://www.snookles.com/slf-blog/2012/01/05/tcp-incast-what-is-it/

[17] Foundation, A. S. (2012). Apache Cassandra. Retrieved from http://cassandra.apache.org/

[18] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News - New York, NY, USA, 33(2), pages 51-59.

[19] Ghaffari, A. (2013). DEbench: A Benchmark Tool for Distributed Erlang. Retrieved from https://github.com/amirghaffari/DEbench

[20] Kargerl, D., Lehmanl, E., Leightonl, T., Levinel, M., Lewinl, D., & Panigrahy, R. (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - ACM New York, NY, USA.

[21] Elie Krevat, Vijay Vasudevan, A. P., & David G. Andersen, Gregory R. Ganger, Garth A. Gibson, S. S. (2007). On Application-level Approaches to Avoiding TCP Throughput Collapse in Cluster-based Storage Systems. PDSW'07 Proceedings of the 2nd international workshop on Petascale data storage New York, NY, USA, pages 1-4.

[22] Lakshman, A., & Malik, P. (2010). Cassandra - A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review - ACM New York, NY, USA, 44(2), pages 35-40.

[23] Lennon, J. (2009). Exploring CouchDB. IBM, Technical library. Retrieved from http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html

[24] Moura e Silva, L., Buyya, R. (1999). High Performance Cluster Computing: Programming and Applications (1st Ed) (Volume 2). Prentice Hall.

[25] Ozsu, M. T., & Valduriez, P. (2011). Principles of Distributed Database Systems (Third Ed). Springer.

[26] SNIC-UPPMAX. (2012). The Kalkyl Cluster. Retrieved from http://www.uppmax.uu.se/the-kalkyl-cluster

[27] Vogels, W. (2008). Eventually Consistent. Queue - Scalable Web Services - ACM New York, NY, USA, pages 14-19.