Master’s Thesis Nr. 28
Systems Group, Department of Computer Science, ETH Zurich

Evaluating Index Architectures in the Cloud

by
Shenji Schäppi

Supervised by
Prof. Donald Kossmann
Simon Loesing

January 2011–July 2011
Abstract

Cloud computing is quickly becoming very popular because it promises virtually unlimited processing power, storage space and high-availability at a fraction of the cost in comparison to having to build and maintain one’s own infrastructure. More and more people are starting to use applications and services in the Cloud and entrust their data to online storage providers. As such, many different Cloud service providers have popped up, each offering slightly different services. These providers use one or more existing Cloud computing storage systems to organize their compute nodes and store data. However, such systems usually do not support indexes. Indexes provide range query support and fast lookups.

In this thesis, we implement various index architectures for the Cloud. We take the B-Link tree, a highly concurrent version of the B+ tree, and adapt it for the Cloud. Instead of simply storing data using a key-value store, which most of the Cloud storage solutions are, the index itself is stored in the Cloud. We adapt the storage architectures Shared-Disk and Shared-Nothing, which are traditionally used in the context of hardware, for Cloud computing. This work also evaluates and analyzes the performance of the index architectures by exploring their parameter spaces and trying to find trends and synergies. Furthermore, we hope that this work provides a base for further fundamental research in distributed indexing architectures. Our benchmarking results show that there are indeed performance advantages when using indexes in the Cloud.
Acknowledgements

I would like to thank my mentor Prof. Donald Kossmann for his continuous support and constantly reminding us of the big picture. I would also like to extend my gratitude to Simon Loesing, who has been of great help during the work of this thesis, from debugging to benchmarking to patiently answering all my questions. Last but not least, I would like to thank my family and friends for their constant support and encouragement.
# Contents

1 Introduction .......................................................... 7  
  1.1 Background & Motivation ........................................... 7  
  1.2 Problem Statement .................................................. 8  
  1.3 Contribution ....................................................... 8  
  1.4 Thesis Structure ................................................... 9  

2 Related Work .......................................................... 11  
  2.1 B-Trees ............................................................ 11  
  2.2 Distributed B-Trees ................................................. 11  
  2.3 Distributed B-Trees in the Cloud ................................. 12  
  2.4 Benchmarking Distributed B-Trees in the Cloud ............... 12  
  2.5 Storage & Architecture ........................................... 13  

3 System Architecture .................................................... 15  
  3.1 Common Ground ..................................................... 15  
    3.1.1 B-Link Tree ................................................... 15  
    3.1.2 Distributed Pointers ......................................... 17  
    3.1.3 Secondary Indexes ............................................. 17  
    3.1.4 Reference Values ............................................. 17  
    3.1.5 Intercomponent Communication ............................... 18  
    3.1.6 Index API ...................................................... 18  
    3.1.7 Index Client ................................................... 19  
  3.2 Storage Architectures ............................................. 19  
    3.2.1 Shared-Disk ................................................... 19  
    3.2.2 Shared-I-Disk ................................................ 21  
    3.2.3 Shared-Nothing .............................................. 22  
  3.3 Persistence Layer ................................................. 22  
    3.3.1 Cloudy ........................................................ 23  
    3.3.2 Berkeley DB .................................................. 24  

4 Implementation Details ............................................... 25  
  4.1 Development Environment ....................................... 25  
  4.2 B-Link Tree ....................................................... 25  
  4.3 Intercomponent Communication .................................. 27  
  4.4 Serialization ..................................................... 27  
  4.5 Clients ............................................................ 28  
    4.5.1 Client API .................................................... 29
Chapter 1

Introduction

1.1 Background & Motivation

Cloud computing has seen a surge in popularity despite the ongoing security concerns. What makes Cloud computing so attractive is that it is simple for the user. The Cloud has the ability to serve clients faster, automatically back-up data and is resilient to failure. On top of that, it is cheap and nowadays, there are options where one gets a bill at the end of the month for whatever one has used, be it, for example, storage space, Cloud services or processing power, just like a water or any other utility bill.

For enterprises, Cloud computing promises lower costs by reducing the total cost of ownership, pay-what-you-use payment policies and no administrative costs while leveraging the processing power of large high-tech datacenters which deliver high performance computing power, automatic backup of data and fault-tolerance at a mouse-click, something not every company can afford to have or even maintain. Additionally, service-level agreements (SLAs) between Cloud computing providers and its clients ensure that client demands such as minimal throughput or maximum latencies are met. Companies can use entire platforms in the Cloud in which software updates, security patches etc. are handled by the provider. This can lead to massive cost savings. For the private user, Cloud computing has made the internet even more useful. As long as there is an internet connection, one can tap into the Cloud. From simply having online storage space which can be accessed from anywhere in the world to complex business applications such as enterprise resource planning or collaborative project management software, a thin client i.e. web browser is usually all that is needed. Since web browsers are not limited to desktop and laptop PCs anymore but are on every smart phone and tablet PC, the user is very mobile and on top of that has a lot of working power at his or her fingertips. This shift of power from the PC back to a conceptually ‘central’ server seems to be the trend. Only this time, this ‘central’ server is larger, more complex and more accessible than before.

It seems that Cloud computing, however, in spite of all the hype around it, does not actually use many new technologies or methods. The Cloud is not much more than a potentially large collection of computers organized in a usually clever way to provide various degrees of scalability, availability and consistency while hooked to the internet to allow global and remote access through some
There are two main ways to organize the data in such compute clusters: decentralized and centralized. Decentralized systems do not have a central (master) node but nodes wanting to join a cluster need some bootstrapping information to get up and running. A popular way to organize the data in such compute clusters is by using dynamic hash tables (DHTs) as is used by systems such as Dynamo [13], Cassandra [9] and Cloudy [22]. On the other hand, centralized systems have a single master and many slave nodes. Bigtable [10] and Microsoft SQL Azure [27] are such examples. In parallel to this, distributed indexes have developed fairly independently. Traditionally, indexes (using B-trees) have been used to speed up and enable certain operations such as reads and scans. It is here where this work has its focus, tying together indexes and data storage systems in a distributed context.

1.2 Problem Statement

The Cloud offers many different services, from ready-to-use software to virtualized clusters with customized operating systems and almost unbounded processing power. However, Cloud storage systems usually only expose a simple interface and do not support index structures. In this work, we shall focus on the data storage aspect and more specifically on storing data in the Cloud using a distributed index. The discussion about Shared-Disk, Shared-Memory and Shared-Nothing architectures [14] has traditionally concentrated on hardware, as in whether or not the hard disk, memory or nothing is shared by multiple processors, respectively. They usually also use specialized interconnection networks for communication among processors or for implementing buses for data transfer. This thesis will look into the Shared-Disk and Shared-Nothing architectures and apply them to the Cloud. Specifically, we want to apply these notions to indexes implemented using B-trees in a distributed computing environment. This means processors are entire compute nodes and the communication network is the Ethernet. The fundamental question(s) we pose but do not pretend to comprehensively answer is: (Why) Should data processing and physical data storage be on the same machine (in line with the latter architecture) or on separated machines (applying the former architecture)?

The two aforementioned architectures are the main index architectures we evaluate in this work. Besides this, we also take a look at variations of said index architectures. The main goal of this thesis is to implement, evaluate and analyze different index architectures and to see if synergies or general trends can be found, keeping the question above in the back of our minds.

1.3 Contribution

The challenge lies in measuring the impact of changing the parameters such as the number of storage nodes, order of the index and payload sizes besides the actual implementation of the distributed index. In particular, we attempt to find out how scalable the various index architectures are with respect to their maximum throughput, how elastic each index architecture is and what performance trends can be seen when varying index parameters. This work
attempts to answer these practical questions and aims to explore the pros and cons of the indexing architectures mentioned before. By providing an extensible implementation of the different index architectures, we hope to also provide impulses for further fundamental research. To the best of our knowledge, no other work has explored the parameter space of such indexes in the Cloud to such an extent.

1.4 Thesis Structure

This thesis is structured as follows.

- **Chapter 1** introduces the reader to the motivation and fundamental questions we pose as well as contribution of this thesis.

- **Chapter 2** looks at related work, putting this work into context. Similarities and differences are highlighted and contributions further clarified.

- **Chapter 3** describes the system architectures of the different indexes and their main components.

- **Chapter 4** goes into the implementation details of the various indexes and their building blocks, pointing out challenges we faced along the way.

- **Chapter 5** provides the methodology used for our benchmarks and experiment details.

- **Chapter 6** shows our experiments, results and analysis of the various index architectures.

- **Chapter 7** presents our conclusions and what is planned for the future.
Chapter 2

Related Work

This chapter provides the context of this work and shows how it fits within the big picture.

2.1 B-Trees

B-trees [5] have become the standard core index data structure not only for file systems [11], but also for popular databases such as Berkeley DB [33] and MySQL [30]. Over the years, there have been many variations of B-trees, one of the more popular one being the B+ tree, which differs from the original B-tree in two main aspects. One is that each leaf node contains a pointer to its right sibling, thereby enabling efficient sequential scans. The other is that there is a clear distinction between index and leaf nodes; index nodes only contain keys and pointers to other nodes whereas leaf nodes contain both keys and their associated data. Again, there are variations where for example, the data of the leaves are themselves pointers to ‘real’ data stored elsewhere. This allows less memory-intensive caching of tree nodes, potentially improving performance.

A natural extension to B-trees is to allow concurrent access. For this, the use of locks seems rather intuitive. However, locks must be placed carefully so that only minimal locking is necessary in order to increase concurrency. The B-Link tree [23] guarantees that at most three locks are needed per process or thread accessing a shared tree. The index structures of this work is based on this variation. The main idea is to have a right-sibling pointer not only at the leaf level, but at all levels of the tree. Using a B-Link tree, read operations do not require any locks at all, further improving concurrency.

2.2 Distributed B-Trees

As distributed computing gained more ground, the idea of replicating index structures across many servers was explored. [25] proposes a lazy index replication approach based on a variation of the B-Link tree, whereby each replicated index grows or shrinks depending on its access patterns. The idea of collecting index information while traversing the remote indexes and then sending them back so local copies can be made is also introduced. However, no experimental
results were published. In our implementation, each B-Link tree can be associated with a cache, effectively resulting in lazy replication. Location-independent identifiers using a global, distributed B-Link tree variation partially based on [25], where individual nodes can be stored on different servers in a hierarchical structure, is discussed in [16]. Besides caching, they propose using replication of individual nodes to improve data locality. We could complement our work with similar ideas at both the index and persistence layers.

2.3 Distributed B-Trees in the Cloud

The work of [1] separates fault tolerance from concurrency control by building a distributed B-tree on top of a distributed data sharing service called Sinfonia [2]. At each client node, a local index copy of a global index is created lazily, which in turn is spread across several servers according to a partitioning scheme. Also, only index nodes are replicated at each client. In line with this idea, we could extend our implementation by having two caches instead of one; one for index nodes and one for leaf nodes. Fault-tolerance is provided by the storage layer and in our case, we use replication. Sinfonia provides so-called minitransactions, which allows atomic distributed updates. By using the service, they move the responsibility of locking to the data persistence level. More recently, [39] use a BATON peer-to-peer overlay network to maintain a global index. This index indexes local B+ trees, which in turn index data shards spread across servers. In other words, they employ a global index of indexes based on the Shared-Nothing architecture. The overlay provides efficient lookups and a tuning algorithm allows dynamic shrinking and expansion of the global index. Our work has a simpler tuning mechanism, namely cache time-to-live and size. It is also possible to change the cache eviction policy to suit the needs of an application, such as using a least-frequently-used (LFU) instead of the default least-recently-used (LRU) policy.

2.4 Benchmarking Distributed B-Trees in the Cloud

The classic benchmarking specifications for databases are given by the Transaction Processing Performance Council (TPC). One of their benchmarks, called TPC-W [37], consists of an e-commerce website and emulated browsers which simulate typical online shopping interactions real users would normally make. Because the benchmark closely represents a real-world e-commerce website, it measures the entire system from application layer down to the storage layer [24]. More recently, [12] (YCSB) provides an open-source microbenchmark framework specifically designed for the Cloud. The benchmark is extensible and provides several predefined workloads. Their experiments compare different Cloud systems in terms of latency versus throughput and elasticity. Besides using our microbenchmarks for testing purposes, our experiments also use an extended version of the YCSB to benchmark our indexes in the Cloud. Additionally, we show results from TPC-W using our index architectures.
2.5 Storage & Architecture

Large distributed compute clusters i.e. the Cloud are nowadays usually organized using DHTs. DHTs such as Chord [35] use consistent hashing [21], which was introduced more than a decade ago. Consistent hashing conceptually partitions a key space in form of a ring. A DHT spreads compute nodes around the ring and each node is responsible for all keys whose hash values lie between the current node and the previous one. Chord provides logarithmic search and insertion times. This in turn formed the basis of many peer-to-peer (P2P) systems, a very popular one being BitTorrent [6]. Also, Cassandra [9] and Cloudy [22] routes keys using such DHTs.

Analogous concepts to Shared-Disk and Shared-Nothing architectures can be found in the context of query processing and are known as Data Shipping (DS) and Query Shipping (QS) [17] respectively. However, in this work, the concepts are applied at a conceptually lower level, namely within the index itself, instead of at the (traditional) client/server level, where client machines interact with server machines on which a database is running. In this work, we use the terms Shared-Disk and Data Shipping as well as Shared-Nothing and Query Shipping interchangeably.
Chapter 3

System Architecture

This chapter describes the architecture of the indexes in detail. In the first section, common components and concepts such as the B-Link tree, which are used across several architectures, are described. In the second section, we describe the two main index architectures and variations thereof. The last section details the concrete persistence layers used in the different architectures.

3.1 Common Ground

Because our system is designed with the principle of high-coupling, low-cohesion as well as reusability in mind, all architectures have common components.

3.1.1 B-Link Tree

The data structure used in the two main index architectures is the B-Link tree. Just like a B+ tree, a B-Link tree is made out of one or more index and leaf nodes. The root node is, upon creation of an index, a leaf node. After the first split, the root node will become an index node. Each node has keys, which are sorted, and values. The values of leaf nodes are the data itself, so each key is associated with a piece of data and we have a one-to-one relationship between keys and values. Note that this data can, in turn, be pointers to other data as is the case when using secondary indexes (Section 3.1.3) or reference values (Section 3.1.4). The values of index nodes contain pointers to its children, which can be index nodes themselves or leaf nodes. Keys in index nodes act as separators for the pointers and therefore the number of keys in an index node is one less than the number of values, if high-keys are not considered (see below). Each tree also has an order which determines the minimum and maximum number of keys and therefore values each tree node can have. Our B-Link tree variation differentiates between leaf and index orders. The root node has a minimum of one key (and two children, if it is also an index node). In general, tree nodes have a minimum of order and a maximum of $2^{\text{order}}$ keys, but index nodes always have one value more than their number of keys. If a node has more than the allowed number of values, a split occurs, starting at a leaf node. Such splits can propagate up to the root node. In a B+ tree, when a node underflows i.e. is smaller than the minimum size after deletions, a rebalancing occurs; keys and
values get shipped to other nodes. If this is not possible, sibling nodes merge. Handling such nodes in this way requires more complex locking and is therefore more prone to deadlocks; a deletion therefore only removes the key and value, without further action.

**Improvements of B-Link Trees over B+ Trees**

The B-Link tree is an extension of the B+ tree in that all levels contain a pointer to its right sibling; a strikingly simple improvement with two major positive consequences. Firstly, read operations do not need to lock any node whatsoever; only modifications such as insertions or deletions need locks. Secondly, during a split, if the parent node, which might be the root, of split children nodes has not been updated yet, read operations will still work because of the right-sibling pointer. This works because when a leaf node splits, it updates its right-sibling pointer to point to the newly created sibling node. If a read request is made for a key which now resides in the newly created sibling node arrives at the parent but before the parent is updated with a key and pointer to its new child, the parent will send the request to the ‘old’ child. This child will then try to find the given key but will fail, and so will hand the search over to its new sibling on the right.

Figure 3.1: An example of a B-Link tree of order 2, created by sequentially inserting the values 0 to 6 - High-keys are written in italics. The vertical grey bars represent the child pointers of index nodes.

Another improvement which the B-Link tree has over the B+ tree is that every leaf and index node contains a so-called high-key. This high-key is the largest key covered by the node. In theory, the high-key of the root node signifies the largest key in the entire tree. In practice, the high-key is lazily updated as leaf nodes and node splits occur. An eager approach would update the keys during a lookup but this would require locking each modified node. Note that it does not matter if a node’s high-key is set to a non-existing value; it is only important to set it to a value greater than all its other keys which includes its child node keys (recursively). The high-key allows more efficient lookups since fast key containment checks can be made. More importantly, it can provide drastic lookup speed improvements especially for read operations during a split as mentioned above, since without it, a lookup for a key which has been moved to the ‘new’ sibling will lead to a fork road. This is because in this case, after
the search ends up at the ‘old’ child, there are two possible pointers to follow:
either its right-sibling pointer or its last value i.e. last child pointer, in case it
is an index node. If the tree is large and this ambiguity occurs high up in the
tree, it is possible that the search will trickle down the tree all the way to the
bottom only to find out that it actually needed to take the ‘other’ path from
the beginning. Of course, one could still continue the search by following the
right-sibling pointers, but this could degrade into a linear search on a linked-list.
A high-key prevents such problems. Figure 3.1 shows an example of a B-Link
tree with both leaf and index order set to 2, built by inserting the keys 0 to
6 in sequence. High-keys are shown in italics at the end of each node and are
separated from the rest of the keys by a solid line. The vertical grey bars in
the index nodes represent the pointers to their child nodes, while leaf nodes in
this example store their data directly as values. As can be seen, even the index
nodes are connected with a right-sibling pointer.

3.1.2 Distributed Pointers
As our index architectures use a key-value-store (KVS) to store the index, we use
a uniform-resource identifier for each node so pointers in our case are URIs. Note
that KVS keys and values should not be confused with index tree node keys and
values. In particular, UUIDs (version 1) are used which are 16 bytes long and
contain the MAC address of the network interface [8]. Because these are globally
unique, a tree can be easily (and lazily) replicated and rebuild from scratch as
long as the root URI is known and access to the KVS is available. In fact,
because of the additional right-sibling pointers of a B-Link tree, it is actually
possible to use or even rebuild the index as long as a node from the left-most edge
of the tree is used and all leaf nodes are still available. Standard index functions
are therefore extended with a URI parameter which is usually used to pass the
root URI (or that of a left-most node as just described) of an index. Although
guaranteed to be unique, UUID version 1 prefixes the computer’s MAC address,
which means that there could be a clustering in the KVS. The Key-Value Store
is the Index index architecture described in Section 3.2.2 uses a custom URI
generator.

3.1.3 Secondary Indexes
Next to primary indexes, our index supports secondary indexes. Secondary index
leaf node values contain one or more primary keys i.e. a set of keys and therefore,
the primary index and keys must exist beforehand. Whether or not a secondary
index value is resolved i.e. looked up in the associated primary index during
read operations or not depends on a switch. This is particularly useful for the
Shared-Nothing architecture (Section 3.2.3), as trees do not necessarily lie on
the same compute node.

3.1.4 Reference Values
Primary index values can also contain reference values (not to be confused with
a secondary index value referencing a primary index key). What this means is
that upon inserting a new value or updating an old to a new value which is
marked as a reference, the actual data is not stored in the leaf node. Instead, a
new URI is generated and associated with the data. This is then stored in the
KVS. The newly created key is stored in lieu of the actual data in the leaf node.
This is especially useful when the data stored at the leaf nodes is very large (see
also Section 3.2.1). Our index does not currently allow secondary index values
to be references.

3.1.5 Intercomponent Communication

The communication between the client and index is achieved by using asyn-
chronous socket communication and message passing. Within the index, all
communication between components such as Federator, Lookup Server, B-Link
Tree Server and KVS (see Chapter 4 for more details) use asynchronous socket
communication as well.

3.1.6 Index API

We assume the underlying KVS to have the basic functions to retrieve (get),
insert (put) and delete data, which we shall collectively call KVS ops. We
subsume the reading functions such as get and getRange under reads and the
modifying functions such as insert and update under mods. We will use client
ops when referring to both. The index provides the following basic interface.

def get(key, uri):
    """The get function, also known as read or lookup operation, navigates down the
tree and returns the value with the given key in case it is found. If, in a pri-
mary index, the value found is a reference, it is resolved and the actual value
is returned. In case of a secondary key, the default behavior is to resolve the
secondary value and return the value found in the primary index."""

def getRange(lowKey, highKey, uri):
    """The getRange function is greedy given a low and high key, which determines the
desired search range; it will try to return as many values as possible within the
range given. For example, given the values [1,2,3,4,5], a getRange search with
the range [-2,3] will return the values [1,2,3]. Using the range [5,6] will return
[5] and using the range [6,10] will return an empty set."""

def insert(key, value, uri):
    """The insert function inserts a key-value pair into the tree. If the key already
exists, the insertion will fail. If the value is marked as a reference, it stores the
value into the KVS and inserts a reference (URI) to that value into the tree with
the given key."""

def update(key, value, newValue, uri):
    """The update function updates an already existing value. If a reference value
is updated, its old associated data is deleted from the store. In the case of

18
secondary indexes, it updates a specific, already existing value in the set of
dvalues at the given key to the given new value.

delete(key, value, uri)

This method deletes an already existing value. If the value at which the key to
delete lies is a reference value, its associated data will be deleted as well. For
secondary indexes, a specific value in the set of values at the key will be deleted.
Currently, no referential integrity checks are made when deleting in primary or
secondary indexes.

3.1.7 Index Client

A client for an index exposes the same interface as described in the previous
section. Figure 3.2 shows how index clients are used by external clients i.e.
other applications wanting to use the index such as benchmarking frameworks.
The index client creates appropriate request messages and sends them to the
index. It then waits for the response before returning it to the external client.
The index part will be shown in more detail in the following sections.

Figure 3.2: The general architecture of the various indexes in the Cloud - Internal
clients, used by external clients, are the gateways to the index.

3.2 Storage Architectures

3.2.1 Shared-Disk

In a Shared-Disk architecture, all processes have a view of the entire data. In
the context of indexing in the Cloud, all nodes are stored in a common space;
we have a pool of nodes where each node’s URI is the key and the node itself
the value of a key-value pair stored in the KVS. To build an index out of this pool, we use so-called B-Link Tree Servers. A B-Link Tree Server contains one or more trees, each with a cache. Each tree initially only has a root node, which has either just been created or was taken from the pool. As searches and insertions are made, new nodes are created and stored in the pool or existing nodes are taken from the pool, building the index as needed in the cache.

Figure 3.3: An example of an index using the Shared-Disk architecture with multiple B-Link Tree Servers (of which one has an additional role as an updater) sharing data.

In this architecture, the B-Link Tree Servers lie on separate machines from the KVS. The B-Link Tree Servers actually transform client requests into KVS instructions; it is the query processor. Note that in our case, although there can be multiple B-Link Tree Servers, all modification requests are handled by a single server; all other servers are used to handle read requests. By having this separation, lost updates are prevented. This updater server’s only difference to the others is its additional role since it too can handle read requests. Figure 3.3 shows an example of an index using a Shared-Disk architecture. The actual index itself is made of the B-Link Tree Servers and the KVS. The index clients send reads to (multiple) B-Link Tree Servers and mods to a single updater B-Link Tree Server.

In respect to this architecture, this means all required nodes not already in cache must be pulled across the network and hence we also call this concept Data Shipping. This is illustrated in the reply after a B-Link Tree Server’s get() in Figure 3.3. Shipping large nodes across the network can be expensive; as a
potential remedy, we look at the Shared-I-Disk architecture described in the following section.

### 3.2.2 Shared-I-Disk

The Shared-I-Disk architecture is a variation of the previously described Shared-Disk architecture. The difference is that we selectively ship leaf node data across the network, instead of shipping the entire node. This is achieved with the use of reference values. If the index has an order of, say 64, and each leaf node value stores 100KB, a full leaf node will contain 12.8MB of data. In the case of the Shared-Disk architecture, this will have a massive performance impact as such large messages take time to transfer over a network and congestion will increase a lot. Instead, by using reference values, a selective data shipping approach is possible. However, the price of this is that an additional request to the store has to be made to retrieve the actual data.

**Key-Value Store is the Index**

Going one step further, one can store the individual values of each node of an index directly as key-value pairs in the KVS. We call this the *Key-Value Store is the Index* (KII) architecture.

![Figure 3.4: An example of an index using the Key-Value Store is the Index architecture - The Federator acts as a kind of query processing middleware between index clients and the KVS.](image)

One can think of this as taking all leaf nodes of a ‘normal’ index and breaking them up into their individual key-value pairs. This also means that there is no concept of tree order like in B-trees; this architecture does not even use
trees for the index. However, this requires encoding the key such that one can retrieve them from the KVS again. This encoding is customizable; one simple scheme is to use a composite key consisting of the root URI and the actual key. For example, the root URI can be used as a prefix or suffix to the given key. Depending on the KVS partitioner, this could lead to different types of clustering of data as is the case if the composite key is made by prefixing the root URI. A hash function, however, should be able to evenly distribute the keys, if so desired. Such composite keys are used to store secondary and reference values as well.

This architecture does not use B-Link Tree Servers, but instead uses a layer between the client and KVS called a Federator, which processes and forwards client requests and returns aggregated results. This new architecture is depicted in figure 3.4. It is possible to develop this idea and have multiple Federators.

Note that this index architecture will incur less lock contention than the others because only individual values need to be locked as opposed to entire tree nodes.

3.2.3 Shared-Nothing

In a Shared-Nothing architecture, data is partitioned among several compute nodes. Since processes do not have a view of the entire data, they need additional information. We employ a similar Federator to the one mentioned in the previous section and a Lookup Server for this purpose. The Federator acts as a middleware which uses the Lookup Server to check on which compute node each key lies and forwards client requests i.e. queries to the appropriate compute nodes, aggregating the results if necessary and returning them back to the client. It also makes secondary index integrity checks.

Each compute node runs its own B-Link Tree Server and indexes the local data; the query processor is now on the same machine as where the data resides. It is the Lookup Server which determines how data is sharded i.e. partitioned across the compute nodes. In this case, tree nodes do not leave their machines, since tree operations are made locally; only the actual node values are returned to the Federator. This concept is therefore also called Query Shipping and is shown in Figure 3.5. The Federator uses the Lookup Server before interacting with the appropriate server. As with KII, multiple Federators could be used.

In the figure, each compute node (cube) contains both a B-Link Tree Server (BLTS) and a separate data store, with the enclosing KVS square merely a conceptual organisation. The interaction between B-Link Tree Server and data store is the same as in the Shared-Disk architecture and has been simplified here for clarity.

3.3 Persistence Layer

The architectures introduced above would not be complete without a description of the storage layers used, which is the purpose of this section.

Note that in theory, the architectures are orthogonal to their persistence layers; any KVS can be used. For instance, we use a hash map-based KVS for testing purposes. In practice, load-balancing, fault-tolerance and number of compute nodes (i.e. cost, maintenance etc.) play major roles in choosing what
Figure 3.5: An example of an index using the Shared-Nothing architecture - The Federator interacts with the Lookup Server in order to find out which compute node, consisting of a B-Link Tree Server and data store, it should contact.

persistence layer to use. More importantly, our index architectures are made to run in the Cloud, and therefore the following choices were made.

3.3.1 Cloudy

For both the Shared-Disk and KII architectures, we use Cloudy [22]. Cloudy is a modular Cloud storage system making it extensible. Components such as the partitioner, routing and consistency protocols are exchangeable, thus allowing the creation of a customized system in terms of availability and consistency with network partitions. Additionally, Cloudy has the ability to repartition the data and load-balance the system not only by redirecting requests and shipping data around but also by adding and removing new compute nodes.

We set the consistency equation \( R + W > N \) introduced in Dynamo [13], which has been adopted by Cloudy, to \( 2 + 2 > 3 \). This means that we use a replication factor of three i.e. every piece of data is (eventually) stored on three nodes and that for both read and write operations, we require success acknowledgements from two different nodes. Furthermore, this means that we can tolerate a single compute node failure. In Figures 3.3 and 3.4, the cubes are used within the KVS to depict compute nodes. Note that interconnecting lines (ring) between said cubes have been left out for clarity. We use Cloudy's Java client to access the basic interface, which is that of a key-value store. Every
B-Link Tree Server can access all the data which Cloudy stores. We configure Cloudy to use Berkeley DB as its local store. Details to the entire system architecture of Cloudy can be found in [29, 34, 26].

3.3.2 Berkeley DB

For the Shared-Nothing architecture, we use Berkeley DB [31] (BDB) for Java with its high-availability (HA) functionality for replication. BDB is an embedded database which means no separate service is needed for it to run: everything can be done by code. It also supports cursors, which allow for efficient scans and transactions. The high-availability feature uses a master-slave approach. Only masters can write to the database, whereas both masters and slaves can read. Also, there can only ever be one master per replication group but many slaves. A replication group is a logical entity, so a physical compute node can take on multiple master and slave roles simultaneously. Since in this architecture, each B-Link Tree Server runs on the same machine as a BDB instance, it only sees a shard of the data. If a master fails and recovers, it will most probably not be the master of its replication group any longer and so cannot process write requests. Currently, such requests are forwarded to the new master. Possible extensions to this are dynamic data shipping of old master data and dynamic repartitioning.

In our case, we use range and modulo partitioning. For range partitioning, every range is associated with one B-Link Tree Server and one BDB instance. For modulo partitioning, which compute node to contact depends on the key and number of compute nodes. The data storage in each compute node in Figure 3.5 represents such a BDB instance. The communication links between the individual compute nodes for replication have been left out to emphasize data sharding. To be consistent with Cloudy’s replication factor, every BDB master has two replicas, which lie on different compute nodes, themselves potentially being a master of a different replication group. The compute nodes can be logically thought of as placed around a ring, with the two replicas of a BDB instance being on the following two compute nodes, wrapping to the beginning when the last node is reached.
Chapter 4

Implementation Details

This chapter delves into the details of implementation. The first section describes the development environment. The following eleven sections explain the workings of the various components making up the index architectures and benchmarks. Finally, we describe some of the bigger obstacles we faced while implementing the index architectures.

4.1 Development Environment

We use Java for implementing the various index variants. For version control, SVN is used. Furthermore, Linux bash and Perl scripts are used in order to simplify managing the different components. We also use the IDE Eclipse’s incremental compiler [15], since it better supports generics. For logging, we use Apache’s log4j.

4.2 B-Link Tree

For both the Shared-Nothing and Shared-Disk architectures, we use a B-Link tree. Both leaf and index nodes are derived from a common tree node parent. Currently, each node stores the entire information of the index, which includes the tree leaf and index order and key type. It also stores primary index information, in case the tree is a secondary index. A viable alternative would be to store this information in a meta data object, which is done for the KII index Architecture. The index uses binary search to look up a given key in a node.

Every node has a list of keys and values. At present, three key types are supported, namely string, long and integer. For index nodes, each value (URI) is a string. For leaf nodes, each value is a set of data. In a primary index, each set contains exactly one element, namely the data associated with the key itself. In a secondary index on the other hand, each set can contain more than one element which are keys of its associated primary index. The data elements at the leaf node level are BT_Values. A BT_Value contains a byte array for the data and a byte to indicate if the data to be stored is a reference value or not.

The original paper uses native memory page locking functions; we use a hash map instead. The modus operandi for the modifying functions insert, update
and **delete** are that they first **find** the corresponding node according to the given key, after which an attempt is made to acquire the lock on the node’s URI. Upon acquisition, the key is searched for again because the previously found node could have been split or otherwise modified in the meanwhile. This re-search is done only horizontally (**scan**) i.e. only sibling-pointers are followed, if necessary. Then, a working copy of the node is made so that modifications in progress do not perturb the underlying index. After modification, one or more (in case of splits) nodes are serialized and stored in the KVS, thereby updating the index.

For the B-Link tree, we would like the root node’s URI to remain constant and not change because of splits. This can lead to an interesting case which can happen (and is a source of deadlocks) and which is not mentioned in [23] when the root node splits after a re-**scan**. For example, imagine having an index which only has one node, the root node, which is at the same time the only node in the tree i.e. a leaf node. This node is full, which means an insert will cause it to split. Additionally, say we have two threads A and B. Now thread A wants to insert a key-value pair. The insert function will determine the current root node to be the node where the insert should be made. If, after this first **find** and before thread A can lock the root node, thread B manages to insert another key-value pair first, causing the root to split, the insert of thread A will fail because the root node is no longer a leaf node. This case is handled by (ab)using Java’s try-catch statement by making use of the fact that casting an index node to a leaf node causes an error: When the exception is thrown, the lock is released and the insert retried. The following code snippet shows how this works:

**Listing 1** (Ab)using Java’s try-catch statement to check if the root node has split after a re-scan.

```java
// Re-scan
findInfo = scan(bt_node, key, false, true, id);
BT_LeafNode<K> bt_leafNode = null;
try {
    // Cast
    bt_leafNode = (BT_LeafNode<K>)findInfo.bt_node;
} catch (ClassCastException e) {
    // Unlock
    ckag.unlock(findInfo.bt_node.getURI());
    // Re-insert
    return insert(key, value, uri, id);
}
```

The original B+ tree supports merging (after **deletes**) and shipping key-value pairs (after **deletes** and **inserts**) with the purpose of balancing the tree. The B-Link tree does not support this kind of rebalancing as it would require more locks, which would defeat its original intent of minimal locking. Instead, a **delete** simply removes the value at a given key, or a value in a set of values in secondary indexes. Empty nodes are not deleted. An extension could be to have a cleaner thread which occasionally scans the index to clean up empty nodes.

26
4.3 Intercomponent Communication

All intercomponent communication is asynchronous and is implemented using Java NIO. We have implemented an Nio Client and an Nio Server which are reused whenever two components need to communicate. This is possible because we work with byte arrays. Most of the time, these byte arrays are serialized messages which represent requests such as get and insert or they are such requests’ return values.

The Nio Server has a single-threaded selector, which is responsible for establishing connections with clients as well as handling read and write requests. This follows the recommendation by [20]. Upon receiving tasks, however, it uses an Executor to handle them by distributing them to multiple RequestWorkers. The same Nio Server uses different workers. For instance, a B-Link Tree Server uses a request worker to handle client ops. A Lookup Server uses a different request worker to handle various lookup requests. These worker threads check the message type and call the corresponding functions. These tasks then call the server’s send function, which queues write operations containing the corresponding return value, which are then handled by the selector thread by sending them back to the corresponding Nio Client. The thread pool size of the Nio Server, determining the number of worker threads available, is configurable in the properties file for each index architecture.

The Nio Client is similar to its server counterpart in that it also has a single-threaded selector, but has to be able to handle simultaneous send requests per client using it. The danger here is that connections get reused by different threads, thus leading to starvation and locks because a response handler of a client op might be waiting for a response, but its associated socket has already been reassigned to another handler. Our implementation provides two ways of dealing with this. One is to use a configurable fixed connection pool, the other is to use a theoretically unlimited connection pool with unique IDs (unique at least for each Nio Server accessing the KVS). The advantage of using a fixed connection pool is that there is no need to generate unique IDs, but there is more overhead in managing sockets which are idle and those which are in use. The pros and cons of using IDs are the inverse of the cons and pros of using connection pooling respectively (and that each communication message contains 4 bytes more because IDs are integers). To support IDs, all index functions take an ID as an additional parameter. When using a fixed connection pool, any number can be passed.

4.4 Serialization

Java has a very convenient built-in serialization feature; one simply has to implement the marker interface Serializable in each class to be serialized and provide a serialization ID for version control. However, this convenience comes at a price: size and speed. Because the standard serialization process not only serializes the current object, but also climbs up the inheritance chain to look for further serialization opportunities, the size of the serialized objects become bloated. To add to this, because reflection is used, speed suffers as well. To counteract these deficiencies, we first tried using the Externalizable interface provided by Java for custom serialization [19]. This turned out to be better,
but we wanted to try to squeeze out more performance. Therefore, we ended up serializing the objects manually; each serializable class contains a `serialize` and `deserialize` method. We also employ byte-packing: Boolean values such as whether or not the index is a primary index and primary key types are packed into a single byte. Sometimes, (de)serializing child objects in the parent object’s (de)serialization function is more size-efficient, since storing redundant information, such as key type information, can be avoided. This dry-squeezing of bytes has been employed only in certain places, as it generally makes the code more unreadable.

The interaction between the external client and internal index client as shown in Figure 3.2 is also realized by message passing. This means that the external client gets the actual values and not a byte array. However, the price is that extra (de)serialization steps must be made.

Table 4.1 shows the sizes of some request messages and their corresponding answers of a primary index with key type `integer` in bytes. The values stored are also integers. The parent `RequestMessage` is six bytes long, containing the message type, the ID as described in the previous section and a boolean value which can be used to suppress return values. Deserializing the URI requires 20 bytes, as an additional 4 bytes to indicate the size of the UUID is needed. Encoding the key type requires 1 byte in addition to the size of the key. This means a `get` message requires a total of 31 bytes and `getRange` message 35 bytes as it requires one key extra to define the range. The `delete` message has the same serialization size as a `get` message, as their contents are the same. The `insert` and `update` messages are bigger due to the fact that, besides the actual value, other information such as whether or not the value to be stored or updated is a reference value needs to be encoded as well. Both `insert` and `update` return a boolean value which requires only 1 byte. A `get`’s response message is slightly bigger than a `delete`’s as it contains the set size as well, which in this example is 1 because a primary index is used. The response message for `getRange` contains the number of sets as well as the individual set elements.

As we have many different types of messages, we use a `Message Type` to distinguish them. This type is used by a common message (de)serialization function in order to choose which message-specific (de)serialization function to call.

### 4.5 Clients

For each index architecture, we implemented one index client. These clients implement an interface called `IControl`, which in turn extends an interface called `IClient`. The latter provides the basic API with functions such as `get`
insert while the former provides functions to start and stop the clients.

All client functions share a similar build in that upon being called, they create request-specific messages, serialize them, send them using an Nio Client, and then wait for a response. Once a response is received, it might be deserialized and further processed before being serialized again and returned to the caller. This is achieved by creating a response handler with each function call. This response handler waits until it has all the data from an Nio Server, whereupon the waiting thread is notified that it can return.

4.5.1 Client API

This subsection details some other functions which are used. Note that not all arguments are shown for simplicity.

scanRange(key, limit, uri)

This method searches for a given key. If not found, it will get the next larger key, and then try to retrieve as many values as possible up to a given limit. This method is only used for benchmarking with YCSB.

contains(pattern, limit, uri)

The contains method returns all records containing a given string pattern up to a given limit. In the B-Link tree, this method works by scanning all leaf nodes beginning from the left-most one of a given index. It is used by the TPC-W benchmark to emulate SQL’s LIKE keyword.

createIndex(uri, makeNew, leaf order, index order, keyType, isPrimary, primaryURI, primaryKeyType)

This function is used to create an index. If no specific URI is given, a random UUID is generated. The makeNew parameter states whether the index should make a new index i.e. create a new root node and store it in the KVS, or just return the given or newly created random URI. This is useful for the B-Link Tree Server (Section 4.6) as trees can be ‘recovered’ by passing a known root URI and setting makeNew to false. Other mandatory parameters are leaf order, index order, keyType and isPrimary which define the tree leaf and index orders, key type (string, long or integer) and whether a primary or secondary index shall be created respectively. Parameters which should be set when creating a secondary index are primaryURI and primaryKeyType which determine the root URI and the primary key type of the associated primary index respectively.

deleteIndex(uri)

As the name implies, this function deletes the index by removing all its nodes given a root URI. It also deletes all reference values, if any, in a primary index.
4.6 B-Link Tree Server

The B-Link Tree Server, which also implements IClient, manages B-Link trees using a map. It is a stand-alone application and uses the Nio Client and Nio Server for passing messages around. It is also possible to predefined primary and secondary indexes in the configuration file. Then, upon starting the server, these trees will be created automatically. Of course, the order of creation plays a role and primary indexes should be specified before secondary indexes. This is especially useful for the Shared-Nothing architecture, where we have a static setup in regards to partitioning. What this also means is that a B-Link Tree Server can be stopped after using it for index creation and restarted with a different configuration to manage the same index by setting the makeNew parameter to false.

4.7 Lookup Server

The Lookup Server is a separate service used by the Shared-Nothing architecture to get the servers responsible for keys according to a partitioning scheme. For inter-application communication, it uses the Nio Client and Nio Server. Its properties file defines what partitioner to use. A default RangePartitioner has been implemented but other partitioners can be created by simply implementing the IPartitioner interface. The TPC-W benchmark for example, uses a modulo partitioner. A possible extension to the currently static setup is to make the Lookup Server dynamic, so that repartitioning can be made on-the-fly. However, this also entails migrating data and is a non-trivial task.

By configuring the Lookup Server such that every server contains the same index, one effectively gets full-replication. However, this can potentially lead to lost updates such as when two concurrent updates are made on the same item but the updates arrive in different order on different compute nodes. This can be solved by synchronizing updates, such as by having a leader or using (global) locks and is planned for the future.

4.8 Federator

The Federator is the middle layer service lying between the clients and the B-Link Tree Servers in the Shared-Nothing and KII index architectures. As it is a separate application, it also uses the Nio Client and Nio Servers for communication. It receives request messages from index clients and processes them with the help of the Lookup Server. In the current implementation, for the Shared-Nothing architecture, if a key of a read request maps to multiple B-Link Tree Servers, a random one is chosen to process the request. Modifying requests are sent to all servers to which the key maps.

When using secondary indexes, multiple hops are made to resolve values. For example, if a get request is made, the Lookup Server is consulted to get the B-Link Tree Server at which the secondary index is located. Once this is found, a request is made to retrieve the set of values (primary keys) associated with the given key. The Federator then iterates over each element in the set and makes requests to the Lookup Server if the key is not already stored in its cache.
The Federator then aggregates the results and sends the response back to the requesting client.

4.9 Cache

The caches are concrete implementations of the AbstractCache class. We use the LRU (least recently used) eviction policy. However, we have also implemented caches with other eviction policies such as LFU (least frequently used). Parameters such as cache size and cache time-to-live (TTL) values are configurable in the index properties file. Note that the cache size determines the number of nodes cacheable and not the actual size in bytes. To decouple the cache from the locking mechanisms of the B-Link tree and in order to prevent multiple threads from retrieving the exact same node from the KVS twice, we read-lock cache gets. The cache also has a cleaner thread, which runs periodically after a configurable amount of time, removing all nodes which have expired i.e. are past their TTL. For B-Link tree caches, the B-Link tree lock map is also used by its cache in order not to evict or clean up locked tree nodes.

Besides B-Link trees, Federators have a cache as well. In the Shared-Nothing architecture, the Federator caches the Lookup Server information and in the KII architecture, the Federator has a separate cache for the meta data and the individual values. As the Shared-Nothing architecture has a local cache per compute node, the data it serves is, in general, fresher than that of the Shared-Disk architecture. This is true once multiple B-Link Tree Servers are used for the Shared-Disk index, because then there will be multiple caches for the same data, and since the caches are not synchronized, data freshness will not be homogeneous. Note that the data freshness in the KII architecture is the same as in the Shared-Nothing architecture, as long as only one Federator and therefore one cache is used.

4.10 Storage Layer

As we would like to plug-in different types of KVSs, we use a IKVS interface which provides the basic operations get, getRange, put and delete. Additionally, the function clear should provide an easy way to clear the KVS if necessary. The general KVS architecture consists of a client and a server. The client is used by the upper layer applications such as Federators and B-Link Tree Servers. The following describes several KVS implementations we use.

4.10.1 Cloudy Key-Value Store

Cloudy, playing the role of the server, provides, among others, a Java client for accessing its data store, which is what the client component CloudKVS uses. As the IKVS interface and that of the Java client are almost overlapping, the implementation is straightforward. Note that we configure Cloudy to use BDB as its back-end store.
4.10.2 In-Memory Store

This KVS uses a hash map and is mainly used for testing purposes, as all data is stored in memory. HashMapKVS provides the interface on the client side, while HashMapKVSServer is the actual stand-alone server. Most KVS functions translate directly to standard hash map ones, making the implementation rather simple.

4.10.3 Berkeley DB

The server component of the BDB KVS, BDBKVSServer, is responsible for setting up the environment, initializing replication and creating a database. These settings are configurable in the corresponding properties file. Since BDB HA is used, we can provide basic failure handling by forwarding write requests on replicas to their master. This is achieved by wrapping each modifying function in a try-catch block and handling the ReplicaWrite exception. Naturally, this only works if a failed master revives after a certain time.

4.10.4 Key-Value Store for KII

IKVS also has a special function called getDPI which uses Cloudy’s data model object (DPI) and is used for the KII architecture only. Specifically, it is used by the functions getRange, scanRange and contains. This is because these three functions use wildcards to get the required key-value pairs. For example, getRange actually ignores the lower and upper bounds when retrieving the values but instead, generates a wildcard string to retrieve all values of an index with a particular URI according to the provided key generator (Section 3.2.2). In the current implementation, these functions bypass the cache. The returned values are then filtered using the range set by the upper and lower bounds. scanRange and contains work similarly, except that the latter uses multiple wildcards. To this end, Cloudy had to be modified to support such wildcards. In general, retrieving all values of an index is a very slow process and as such, these functions should be used sparingly in this architecture.

4.11 Cache-KVS Aggregator

The aggregator brings cache and KVS together and is used by each B-Link tree for getting and putting tree nodes. By using the aggregator, get requests will first check to see if the requested node is already in the cache and still fresh. If so, no message has to be sent across the network and the node from the cache is used. Otherwise, a get request is sent to the KVS and the cache is updated upon receiving the requested node. If a B-Link tree wants to write back a modified node, it is first put into the cache and then into the KVS.

Reference values (Section 3.1.4) are handled in a special way. Only resolved reference values are stored in the nodes in the cache. For instance, let us say we have a leaf node with only reference values i.e. URIs and an empty cache. A request made for a key which lies in that leaf node will lead to it being stored in cache. As part of the request, the reference value is resolved and also stored in
cache. Specifically, in the now cached node, only the value holding the reference URI is replaced with the actual data.

4.12 Benchmark

In this section, we detail two benchmarking frameworks which were used in this work. Details on the benchmarking methodology can be found in Chapter 5.

4.12.1 In-house

For initial tests, we used a custom-made extensible benchmarking framework. This framework uses loads as its underlying work unit. Each load is a specialization of a parent AbstractLoad, and represents a unit of work such as a get, input or update request. We have also implemented random loads, which, in case of get requests for example, generates a random key instead of using a fixed one. Each load measures the time it takes for a call to a function of an index client to return. There is also a composite load, which, as the name implies, can contain other loads, including other composite loads. Each composite load is configurable to run multiple times. To make workloads (a bundle of basic loads) for this benchmark, we have load generating functions which fill up composite loads with various kinds of elementary loads. It is therefore easy to add other kinds of loads which have e.g. different key distributions or simulate hotspots. What workloads to use and how to generate the different workloads is specified in individual benchmark configuration files; mixes of different workloads as well as periodic running of workloads are configurable. To modify throughput, the number of threads can be adjusted per workload.

For each architecture, a so-called Benchmark Driver instantiates the appropriate client and calls the corresponding functions based on the workloads. The gathered times are either printed out to standard output or to a file depending on the specified ITimePrinter. It is also possible to send the collected time information to a separate server. To synchronize multiple benchmark drivers, a future start date and time for each individual workload can be set. By placing multiple benchmark configuration files in a folder specified in the benchmark driver configuration file, multiple benchmarks can be run automatically. A Populator application has been created to easily fill the KVS with data. It is also noteworthy that in each benchmark configuration file, population (separate from the previously mentioned populator) and benchmark warm-up parameters can be specified.

4.12.2 YCSB

The Yahoo! Cloud Serving Benchmark provides default workloads for benchmarking performance and scalability. By being able to specify a desired throughput, latency-throughput curves can be made easily. We use a customized workload for our benchmarks. This is because our indexes do not have the concept of rows and fields (although it could be modeled). In particular, we directly use a string version of the generated key instead of appending the predefined string ‘user’ to each key, as is done in the default CoreWorkload.
For each index architecture client, we create a class which extends the provided DB class. When running the benchmark, the YCSB client creates threads. Each such thread in turn instantiates such a class. To this end, the provided default workload files have been enhanced with properties necessary for instantiating the corresponding clients. Furthermore, a RANDOM property allows random assignments of B-Link Tree Servers for reads to different threads when using multiple servers while accessing the same tree for benchmarking; the updater B-Link Tree Server remains fixed, however. An additional property called TABLE_NAMES determines if, in case multiple different trees are used, the tree root URIs i.e. table names are assigned randomly or not. Using both aforementioned properties, it is thus possible to make each thread contact a random B-Link Tree Server and use a random B-Link tree on that server. This assignment is fixed for the life of each thread.

4.13 Challenges

One of the major challenges during implementation, as is usual in the software development world, is that requirements change. The code was refactored several times when introducing new features such as support for references and secondary indexes. Although refactoring usually involves a lot of changes, we managed to piggy-back cleaning up the code on top of adding new features. Actually implementing the architectures, as was anticipated from the beginning, took up most of the time, leaving less room for benchmarking. Nonetheless a solid index layer has been build on top of Cloudy and other KVS’s, giving future experiments a kick-start.

An early obstacle were deadlocks. Although [23] does provide pseudo code, challenges such as the one mentioned in Section 4.2 on B-Link trees, the fact that dereferencing in a low-level programming language such as C meant making explicit copies in a high-level programming language such as Java and having to use lock maps were not so obvious in the beginning.

We also had to modify the Nio Client in Cloudy to support handling multiple threads by associating each socket channel with a byte buffer. Additionally, Cloudy’s BDBStorageEngine was fixed, as it deleted records before inserting them.

Early on, the Java language’s generics feature seemed to be a promising solution for creating reusable, flexible code. Using generics, the index should have been able to support any type, or so we thought. The limitations of generics soon became apparent. Explicit casts still had to be made in many places. In the end, generics were kept to a minimum and key types which the index supported were defined and passed as parameters explicitly. This involved creating supporting functions to transform e.g. secondary index values, which are byte arrays, into primary keys, which currently can be of type string, long or integer. By doing this, the code was simplified and became more readable. The drawback is that the external client must make sure that the parameters passed when calling functions are of the types corresponding to the primary and secondary key types as specified during tree creation, or else runtime errors will occur.
Chapter 5

Benchmarking Methodology

This chapter describes the benchmarking methodology used for our experiments. The first three sections describe the benchmarks used, what metrics were considered as well as in which environment we ran our experiments. The last section details where the focus of our experiments was and shows our hypotheses regarding several experiments.

5.1 Benchmarks Used

For our experiments, we used two benchmarks: YCSB and TPC-W. This section describes the two benchmarks and the way they were used.

5.1.1 YCSB

YCSB provides a framework for measuring latency and throughput in a relatively easy manner. The accompanying paper also provides some benchmarking results of other systems, making it interesting to make initial comparisons, although we are not trying to compete with any system, but rather are evaluating different index architectures in the Cloud. The benchmark provides four different functions (read, update, scan and insert) which are each called with a definable frequency, whereby the sum of the frequency proportions of each function must add up to 1. The keys generated during a benchmark run are based on a given distribution (uniform, Latest, Zipfian). The Latest distribution ensures that newly inserted keys are the most popular; they are generated most frequently. The Zipfian distribution models the fact that some keys may be very popular and remain so regardless of new insertions. The provided default workloads differ in the allocated frequency proportions and key request distributions. A command switch outputs a summary line to standard error every 10 seconds which allows to see if a running benchmark is still alive.

Workload

For our experiments, we use the default provided workloads A, B and E, which all use a Zipfian distribution per default. Workload A is update-heavy with a
read-update frequency ratio of 0.5-0.5. Workload B is read-heavy with a read-update frequency ratio of 0.95-0.05 while workload E has a scan-insert frequency ratio of 0.95-0.05 and a default scan length (number of records to fetch) of 100. For our experiments, workload E is modified to scan only i.e. scan frequency ratio of 1 because the QS architecture uses a static partitioning based on the number of inserts during loading and so additional inserts beyond the initial population amount will fail to map to a server during a lookup.

Each workload also has the properties recordcount and operationcount, which tell the benchmark how many records to insert initially and should be considered (used for initializing key generators, amongst others), and how many actual operations to execute respectively. When using the workload in loading mode, as opposed to the benchmarking transaction mode, both properties should be equal. We ensured that almost all experiments ran for more than a minute, as this is especially important for the cache TTL parameter experiment. Note that [12] does not mention the runtime of each experiment.

**Experiment Setup**

The system under tests (SUT) in our benchmarks are the indexes including the index clients as shown in Figures 3.3, 3.4 and 3.5. The external client is the YCSB client with each thread instantiating an index client. We remind the reader that our benchmarks use the index client to access the actual index (Section 4.4). For our benchmarks, we need to define default values for each parameter we plan on varying, thus allowing us to make fair comparisons within and among the different architectures. This is shown in Table 5.1. In all of our benchmarks, both leaf and index orders are equal and we shall refer to both as tree order. The parameters we explore per architecture are the number of storage nodes (which we simply call servers) and trees, tree order, payload size and cache time-to-live (TTL).

<table>
<thead>
<tr>
<th># Storage Nodes</th>
<th># Trees</th>
<th>Tree Order</th>
<th>Payload Size</th>
<th>Cache TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>16</td>
<td>100B</td>
<td>60s</td>
</tr>
</tbody>
</table>

Table 5.1: Default parameter values used as a base for all experiments.

For the DS architecture, we additionally vary the number of tree nodes i.e. B-Link Tree Servers, using 3 as the default. Note that this parameter is neither applicable to the QS nor KII architectures.

The payload size is the size of each value associated with a key in a leaf node of a primary index for the Data Shipping and Query Shipping architectures, and the size of each value per key-value-pair in the KII index architecture. The cache size is set to 10'000 nodes (Section 4.9) for each B-Link Tree Server. The default load is 100'000 * 2 * TreeOrder inserts per tree, which means that at least 100'000 leaf nodes and therefore more than 100'000 nodes will be created in total, implying that less than 10% of the total number of nodes will be able to fit in the cache. Using the default parameters, this amounts to 3’200’000 inserts. As such, the maximum (minimum) height for a B-tree and therefore B-Link tree is given by the formula \( \lceil \log_t n \rceil \) (\( \lceil \log_2 n \rceil \)), where \( t \) is the tree order and \( n \) the number of values. This concretely means that the tree is at most 6 but at least 5 levels high. This also means that in the case of KII, we use a
cache size of 320,000 values for the Federator, since individual values are stored and not entire tree nodes. Whether or not this is fair with respect to the other two architectures is arguable and the interested reader can find a custom value or formula if desired.

It should be noted that although the potential maximum amount of values stored in cache is the same, the cache (and locks) used by KII and the other architectures are different (individual values vs. tree nodes) and could have a different impact on performance. The cache cleaner thread runs every 30s and each Nio Server is configured to have a fixed thread pool size of 400. Both Nio Server and Nio Client are set to use IDs (Section 4.3). For each of the experiments, we use a YCSB client to run the workloads while increasing the target throughput until it cannot be reached anymore.

In our experiments, we reload the database each time the number of storage nodes, tree order and payload size changes. As loading can take a long time (several hours or more for tens of millions of inserts using the default parameter settings), we have implemented a feature we call flush mode which allows inserted nodes to be held in cache before being flushed i.e. written to the store after a specified interval. As this feature should not be used during benchmarking, we exploit the ability to create or recover trees by configuring the B-Link Tree Servers appropriately (Section 4.6); after loading is complete, we restart the B-Link Tree Servers without the flushing feature.

All indexes in the experiments are configured to use integers as the key type. When using Cloudy, only one and the same seed is specified in each local configuration file. Each YCSB workload starts with a target throughput of 1000 and we increase it in steps of 1000 until it can no longer be reached. More details of the individual architectures follow.

DS

For the Shared-Data architecture, the tree is duplicated on each tree node used. The steps for setting up each benchmark with this architecture is as follows:

1. Start Cloudy and make sure it is running by e.g. checking its web interface.
2. Start a B-Link Tree Server for updating, optionally configured to use flush mode.
3. Run a YCSB client in load mode.
4. Restart the updater B-Link Tree Server without flush mode if it was used during loading.
5. Start one or more B-Link Tree Servers for read operations
6. Run a YCSB client in transaction mode.
7. Collect results.

QS

For the Shared-Nothing architecture, we evenly spread the data across the number of servers used. Because this is currently done in a static manner, the
keys cannot be hashed before being inserted and so the workload parameter
\texttt{insertorder} is set to \textit{ordered}. So if three servers are used, then, based on the
keys, the first third of the data is stored on the first, the second third on the
second and the last third on the last server. We use one Federator and one
Lookup Server, each running on separate compute nodes. The steps for setting
up each benchmark with this architecture is as follows:

1. Start all BDB instances and wait for all replica groups to be established.
2. Start all B-Link Tree Server instances, optionally using flush mode, on the
corresponding compute nodes where BDB instances are running.
3. Start the Lookup Server.
4. Start the QS Federator.
5. Run a YCSB client in load mode.
6. Restart all B-Link Tree Server instances without flush mode if it was used
during loading.
7. Run a YCSB client in transaction mode.
8. Collect results.

\textbf{KII}

The steps for setting up each benchmark with the KII index architecture is a
mix of the previous two setups. We use \texttt{URIKeyGenerator} as the default key
generator, which prefixes the root URI to each key.

1. Start Cloudy and make sure it is running by e.g. checking its web interface.
2. Start the KII Federator.
3. Run a YCSB client in load mode.
4. Run a YCSB client in transaction mode.
5. Collect results.

\section*{5.1.2 TPC-W}

TPC-W is the quasi de facto standard when it comes to benchmarking web
applications having a database backend and involving thousands of clients. It
simulates an e-commerce website and uses so-called emulated browsers (EB),
which represent clients interacting with the website, be it browsing or making
purchases, to provide the load on the SUT. The benchmark specification de-
scribes the architecture of the system, the SQL statements that should be used
as well as benchmark specific metrics in full detail \cite{37}.

In the following subsections, we will state the workload and settings used
for this benchmark. We use a modified TPC-W benchmark customized for
benchmarking Cloud systems and measure the WIPS (Web Interactions Per
Second) given a number of EBs. However, going into detail as to what changes
were made, how each database attribute relates to the other and the system architecture of the benchmark itself is beyond the scope of this work. An in-depth explanation and details can be found in [24].

Workload

There are three mixes in TPC-W: browsing, ordering and shopping mix. A mix in TPC-W is a combination of browse and buy interactions that an EB can make. The transitions to take from one state to another is defined in a mix-specific transition matrix made out of probabilities. For the TPC-W benchmarks, we use the ordering mix as our workload.

Experiment Setup

The data size including indexes is around 1GB using 10'000 items. For the QS architecture, we fully replicate (Section 4.7) the database entries Items, Authors and Cartlines and partition by custId and cartId using a modulo partitioner. Nio Clients are set to use connection pooling with a pool size of 100. For DS, each B-Link Tree Server’s Nio Server has a connection pool size of 32 and that of QS’s Federator 400. Also, one Federator and a Lookup Server are used. For both DS and QS architectures, we vary the amount of B-Link Tree Servers and B-Link Tree Server cum BDB instances respectively.

We use load generating servers with a capacity of 5000 EBs each, whereby each EB corresponds to one thread. The total load of the system increases by 1 EB every 0.4 seconds (derived from the time it would take to start 9000 EBs in an hour).

5.2 Benchmark Metrics

In the following section, we will describe some general benchmark metrics and relate it to our experiments. Although YCSB introduces the four benchmark tiers performance, scalability, availability and replication, the latter two have yet to be implemented.

5.2.1 Performance

The classic benchmark metric, performance in our case is measured in terms of throughput and latency. Because YCSB allows defining a target throughput, the corresponding latency can be measured easily. We measure performance in all experiments to see how latency changes as target throughput is increased until the peak throughput is reached.

5.2.2 Scalability

This metric measures how well a system can scale i.e. whether by having additional machines in the system, higher (peak) throughput or lower latencies can be achieved given enough load. Together with the performance metric, our experiments measure scalability by using the default parameter settings but increasing the number of storage nodes. A scalable system can achieve higher
maximum throughput given a fixed latency with an increased load (or lower minimum latencies given a fixed target throughput) by adding more nodes. In our experiments, finding a limit or at least a trend is of particular interest.

5.2.3 Elasticity

Elasticity can be thought of as the dynamic counterpart of scalability. While the latter measures the peak throughput of static systems of increasing sizes, elasticity shows how the system ‘behaves’ when a new server is added to a running system. Additionally, the load can be increased proportionally when adding such a new server. Of interest is how fast the system’s throughput stabilizes i.e. how long it takes until it shows equal behavior as if one had statically configured the system with the new amount of nodes from the beginning as well as whether similar peak throughput in an analogous scalability benchmark can be reached.

YCSB’s time-series functionality can be used for this, which periodically outputs the throughput of the benchmark over its life time. The more elastic a system is, the faster it stabilizes and returns to performing like in a static setup. One way to measure this is by adding a new node, keeping the target throughput fixed and then looking at how long the system takes to stabilize while also measuring latency.

5.2.4 Availability

Generally, availability measures how well a system can handle changes such as failures (fault-tolerance) or a sudden increase in the number of requests. One of the simplest way to measure this is to have a running system, kill a node, and see how the system responds (in terms of throughput and latency). More concretely, one could use YCSB’s time-series functionality to run a benchmark and stop a sever after a specified amount of time. However, there are several aspects to this, such as whether something like an ‘inverse elasticity’ is measured to see how the system behaves when a node is removed or even combine availability and elasticity to see if a node failure followed by a node recovery behaves in any way different than the standard stable case. As both Cloudy and BDB are configurable as to whether log files are flushed to disk on writes or not, this can be another possible point of measurement. Also, speed of automatic recovery is another aspect.

5.2.5 Replication

Like YCSB’s tiers, we consider replication as a separate measure. However, it is also possible to instead, consider replication as part of availability and therefore also fault-tolerance. Initial measurements using BDB with replication in the Shared-Nothing architecture have shown that replication can dampen throughput substantially but further work in this direction is needed to draw more solid conclusions. Using the current stand of YCSB, we could measure the performance of replication by varying the replication factor and running the benchmarks with the default settings.
5.2.6 Consistency & Durability

The current implementation of the three index architectures guarantees that updates are never lost. However, using the Shared-Nothing architecture with full-replication could lead to lost updates (Section 4.7), but the benchmarks in this work do not employ full-replication, except for those using TPC-W. Cloudy uses eventual consistency, as it is partly based on Cassandra. Eventual consistency means that updates will sooner or later be propagated to its replicas. Cloudy is configured to use a quorum protocol, which waits for a majority of nodes when reading or writing, ensuring the equation in Section 3.3.1 is fulfilled.

BDB with replication is tunable in three aspects in terms of durability. First, the master of a replication group can be configured to either write to and flush the write-logs upon transaction commit to disk synchronously, write to the logs but not flush it on commit or to neither write nor flush the logs synchronously. The second option is somewhere in between the other two in terms of durability; the write-logs are saved in the file system buffers, preventing data loss during an application failure but not when hardware fails. Second, each replica can send an acknowledgement after flushing writes to disk synchronously or the flushing can be done asynchronously. Third, the acknowledgement policy of replicas on transaction commit can be set to either a simple majority (as is used for the Shared Nothing architecture benchmarks), none or all. For more information, see [32].

For our benchmarks, we set BDB and Cloudy (because we configure Cloudy such that each node uses BDB without high-availability as its local store) to write to and flush write logs asynchronously. Furthermore, the replicas do not have to flush their logs before sending an acknowledgement.

Provided that there are no failures, the TPC-W implementation using the index architectures provides monotonicity for individual operations. This is because the four client-centric consistency levels monotonic reads, monotonic writes, read your writes and write follows read as described in [36] are fulfilled by virtue of using a single updater - an updater B-Link Tree Server in the case of the DS index architecture and a Federator in the cases of the QS and KII index architectures - and redirecting reads requiring the most recent updates to the updater.

To provide relational database-like transactional support and stronger consistency guarantees in general, especially when using multiple B-Link Tree Servers, we can use queues following ideas presented in [7]. This can also be used to synchronize caches, thus preventing lost updates when using full-replication and providing monotonicity or even atomicity for the index architectures, independent of the benchmark used.

5.2.7 Cost

A metric which is very important especially in Cloud computing is cost. To measure this, one could run the both the YCSB and TPC-W benchmarks using an available utility computing service such as Amazon’s EC2 [3] or Microsoft’s Windows Azure [28], as they provide statistics and a bill at the end of the month.
5.3 Benchmark Environment

All benchmarks were run using the following environment: On the software side, we used Oracle’s JDK 1.6.0_25 for running Java and Eclipse Compiler version 3.6.2 for compiling. On the hardware side, we used a server cluster consisting of 64-bit 16 Core 2.26 GHz Intel Xeon CPU machines with 32GB of RAM running Ubuntu 10.4.2. Additionally, gigabit Ethernet was used, although the bandwidth is not exclusive to the cluster, which could, depending on network traffic, skew results.

5.4 Performance Focus

Our benchmarks focus on scalability and parameter space exploration in regards to throughput and latency. In addition, scans are also looked at. The other metrics are slated for future work. To form a basis of comparison among the different architectures for scalability, we compare peak throughput at a fixed maximum latency. For the parameter space exploration experiments, the default parameters are used while varying one parameter as described in Section 5.1.1.

5.4.1 Parameter Space Hypotheses

This subsection will present our hypotheses of the impact of changing individual parameters on performance, in particular on throughput. Note that as latency is closely tied to throughput, reversing the graphs of either should show the same trends. As YCSB allows to specify a target throughput, fixing a maximum throughput and showing the latency-parameter plot shows us more accurate results than vice-versa.

Number of Servers & Scalability

In theory, adding compute nodes to the storage layers of the different index architectures should increase performance. For DS and KII, this is because the data is more load-balanced i.e. less clustering and hot spots and the replication inequality of Cloudy can be more easily fulfilled since there are more servers. For QS, this is because the trees are smaller per compute node, which means that less nodes need to be fetched thus reducing tree traversal time. In addition, QS has the advantage of being able to have more parallel updates which implies less lock contention, as each tree on each compute node is independent from each other.

Because we use BDB HA, replication is static and therefore this overhead should remain relatively constant as the number of servers increase. In practice, the communication overhead will brake performance, especially for the architectures using Cloudy, since partitioning and load-balancing information as well as nodes need to be passed around. Also, the Nio Server (Section 4.3) uses a single-threaded selector, which might become a bottleneck if there are a lot of threads.

Figure 5.1 shows the theoretical and practical throughput trend as the number of servers increases. Note that there could be an optimum, or the performance could even out, depending on parameter settings. However, we suspect
that even with a constant load, maintaining a constant ‘peak’ performance with increasing number of servers will be difficult due to increasing communication overhead, especially for DS and KII. QS should scale until the Federator or the network bandwidth becomes the bottleneck.

**Tree Order**

As tree order increases, update performance will decrease for all architectures because tree nodes become bigger, thus needing more time to be (de)serialized and possibly shipped across the network. Also, there will be more lock contention as an updating thread will lock an entire (large) node. However, read performance might increase up to a certain point if caches are used, especially for sequential access. Figure 5.2 shows how throughput might change as the tree order increases. As the tree height increases exponentially when the tree order decreases, there should be a point where performance will drop due to longer tree traversal time. In addition, if the cache size i.e. number of nodes cacheable is constant, the probability that nodes need to be evicted increases, further negatively affecting performance. The optimum will differ for each index architecture, and potentially for each operation as well.

**Payload Size**

The effect on performance in relation to payload size should be quite straightforward: The performance of using a larger payload will be worse than when using a smaller payload.

Figure 5.3 shows how throughput should perform as payload size increases. Also, one cannot realistically expect that using a e.g. 10x smaller payload will result in a performance gain by that factor, since although using a smaller payload means more smaller network packets are sent, the size of, for example, one network packet containing a 20B index payload is not the same as the total
Figure 5.2: The development of throughput with increasing tree order.

Figure 5.3: The development of throughput with increasing payload size.
size of two network packets containing a 10B index payload each. Additionally, nodes need to be evicted more frequently as cache size is limited, preventing such a 1:1 performance increase. Nonetheless, a linear drop in throughput can be expected.

**Cache TTL**

As the cache TTL increases, so will the performance of the various index architectures up to the point where the cache TTL equals the experiment runtime. At this point, further increasing the cache TTL should not have any effect on performance, as nodes are evicted only if the cache is full and because the cleaner will have nothing to clean up. The decrease in performance when reducing the cache TTL (lower than the benchmark runtime) comes from the increased number of node retrievals from storage.

![Figure 5.4: The development of throughput and data freshness with increasing cache TTL.](image)

Figure 5.4 plots this hypothesis, with the dashed line showing the throughput limit determined by the benchmark runtime. When not using a cache i.e. when cache TTL is 0, performance for all index architectures will drop, but DS will be especially effected, because it needs to always ship a relatively large amount of data across the network. As already mentioned in Section 4.9, data freshness decreases as cache TTL increases (depicted by the red dotted line in the figure).

**Number of Tree Nodes**

In DS, increasing the number of tree nodes will likely increase the performance for reads because more parallel access to Cloudy can be attained. Note that although the total cache size also increases, the caches are not synchronized and so the amount of old data increases with the number of tree nodes. Update performance should remain relatively stable if the offered throughput remains constant, as still only one updater B-Link Tree Server is used. Using a read intensive workload, the B-Link Tree Servers (for reads) will likely become the bottleneck if only a small number is used, as each tree node is more loaded.
However, using an update intensive workload, the updater B-Link Tree Servers will eventually become the bottleneck. We hypothesize that there will be a scaling limit, just like in Figure 5.1, as inter-node communication in and the number of parallel connections to the KVS increases.

**Number of Trees**

For the KII index architecture, increasing the number of trees while keeping a constant load should not make much of a difference on performance, because it just means that more nodes are added to the storage pool. Of course, filling the storage system with many (hundreds of millions) of values will slow down the system as retrieval and storage take longer. For both the DS and QS architectures, adding trees might increase update performance because there would be less index lock contention since modifying operations are more spread out among the trees. As for read performance, although there will be less cache lock contention since each index will have their own cache, the sizes of the cache would have to be inversely proportional to the number of trees i.e. smaller, which in turn means that more nodes have to be fetched from the store. This increase of data transfer will most likely outweigh the benefit of having less cache lock contention and therefore lead to a decrease in read performance, asymptotically nearing the cache-less performance (i.e. cache TTL equals 0) as depicted in Figure 5.5.

![Figure 5.5: The development of read and update throughput with increasing number of trees.](image)

46
Chapter 6

Experiments & Results

This chapter shows some fruits of our labour. The first section presents our findings while exploring the parameter space. The following section looks at other benchmarks such as scans and TPC-W using the index architectures. Finally, we mention some optimizations and limitations we faced. Also note that we use DS, QS and KII to mean both the concepts and the index architectures themselves.

6.1 Parameter Space Exploration

In this section, we present our YCSB benchmark results of the different parameters. Each experiment varies one parameter while using the default parameter settings as described in Section 5.1.1 for the others. Because of mutual influence i.e. reads affecting updates and vice-versa, the graphs for read and update performance per workload look very similar and so we usually only show the update and read operation graphs of workload A and B respectively.

6.1.1 Number of Servers & Scalability

Experiment

We benchmarked the three architectures while varying the number of storage nodes from 3 to 7 with a step of 1. This is also our scalability experiment. We hypothesized that QS scales better than DS and KII, because both DS and KII have more inter-node communication in the persistence layer and QS has increased parallelism (Section 5.4.1).

Results

**DS** In general, the trend is that as the number of servers increases, the maximum throughput increases as well - up to a certain point. This point is at 6 servers in our experiments and can clearly be seen when using workload A (Figure 6.1a): Higher throughput and lower latencies can be achieved using 6 servers than with 7. One possible explanation for this is that, as hypothesized, the communication overhead of Cloudy starts to slow things down. Interestingly, the read performance plot in Figure 6.1b using workload B does not show such an obvious difference. As workload B has much less writes and writes are slow
in comparison to reads because of an extra hop needed to write back to store (which includes (de)serialization), it could be that the communication overhead further emphasizes this difference.

![Figure 6.1: Varying the number of servers for the DS index.](image)

(a) Workload A - updates  
(b) Workload B - reads

Figure 6.1: Varying the number of servers for the DS index.

Figure 6.2 shows the effects of scaling on throughput more clearly, namely the maximum throughput (with the given granularity of 1000 operations) at a maximum allowable latency. For workload A’s read and update operations, the maximum latency is set to 10ms. For workload B, 5ms is used for both operations. These numbers were chosen such that they lie within the minimum and maximum latencies of each experiment per architecture. As observed previously, DS seems to scale well up to 6 servers. The reason using 4 or 5 servers does not seem to make any difference in terms of throughput when using workload A (Figure 6.2a) is probably due to the relatively large granularity, which means less accurate plots (see also Section 5.4.1).

![Figure 6.2: Throughput trend for DS while increasing number of servers with fixed maximum latencies (in parenthesis).](image)

(a) Workload A  
(b) Workload B

Figure 6.2: Throughput trend for DS while increasing number of servers with fixed maximum latencies (in parenthesis).

**QS** In contrast to DS, QS seems to be able to scale beyond 6 servers as can be seen in Figure 6.3. Further experiments with more servers are needed to see just how far this index architecture can scale. As with DS, the difference between using 6 and 7 servers is more pronounced for workload A (Figure 6.3a) than for workload B (Figure 6.3b). In this architecture, however, using 7 servers gives lower latencies and higher throughput, which is especially noticeable for the
update heavy workload A, probably because of increased concurrent updates and smaller trees. For workload B, the scaling out effect is not as obvious, perhaps indicating a limit of BDB’s replication feature, that the Federator has become the bottleneck or that the network bandwidth limit has been reached. The spike when using workload B and 4 servers is most probably an outlier due to, for instance, a sudden short burst of network traffic, a JVM garbage collection run or an increase in index node retrievals from storage.

(a) Workload A - updates
(b) Workload B - reads

Figure 6.3: Varying the number of servers for the QS index.

As with DS, Figure 6.4 shows how throughput develops as the number of servers increases. For workload A’s read and update operations, 2ms and 6ms were used for the maximum latency respectively. Workload B used 2ms for both operations. As mentioned before, QS scales beyond 7 servers, which is especially obvious with workload A (linear scale out).

(a) Workload A
(b) Workload B

Figure 6.4: Throughput trend for QS while increasing number of servers with fixed maximum latencies (in parenthesis).

KII The plots of KII in Figure 6.5 show that there is a big performance benefit when scaling from 4 to 5 servers, after which further scaling only brings minimal differences to both workloads. This could be because inter-node communication at the storage layer (Cloudy) starts to dominate. Notice that the plots using 3 and 4 servers in Figure 6.5b show a belly shaped curve, which could indicate that the cache is still cold or that there are more evictions happening in the cache.
Although being different architectures, KII scales to about 6 servers as well, just like DS. This can be seen in Figure 6.6, which shows the development of throughput using a fixed maximum latency: For workload A’s read and update operations, 3ms and 4ms is used respectively and for workload B, 2ms is used for the read and 3ms for the update operation. This could again imply that the underlying storage system, Cloudy, scales well up to around 6 servers.

**Synthesis**  
Figure 6.7 compares the three architectures using 3 and 6 servers. Comparing Figures 6.7a with 6.7c, it is interesting to see how, as the number of servers increases, for workload A, QS’s update latency trend approaches that of KII, while for workload B (Figures 6.7b and 6.7d), KII’s read latency trend comes close to that of QS.

Sometimes, it is of interest not only to see what the peak throughput given a certain latency limit is, but also the inverse; given a certain throughput, what latency can be expected? Figure 6.8 shows exactly this. Its Subfigures (a) and (b) plot the latency with a maximum throughput of 5000 ops/s and 8000 ops/s respectively with increasing number of servers. As expected, similar observations regarding scalability can be made here i.e. best performance at 6 servers for DS, and that QS seems to scale, but KII’s performance stagnates after 5 servers.

Globally speaking, under the settings described in the previous chapter, QS
Figure 6.7: Comparing the performance of the different architectures using 3 and 6 servers.

has the better read while KII has the better write performance. However, this could also be due to the fact that the caches they use are different (Section 5.1.1); locally (i.e. on a single machine) shipping large nodes (QS) might be less efficient than shipping small chunks of data across the network (KII). Comparing the fixed maximum latencies plots with the graph in Section 5.4.1, one can see that these results support our hypothesis.

Figure 6.8: Latency trend for the three index architectures while increasing number of servers with fixed maximum throughputs.
6.1.2 Tree Order

Experiment

For this benchmark, we varied the B-Link tree order to assume the values 8, 16, 32, 64 and 128. Note that we only benchmarked the DS and QS architectures, as KII does not use trees. We hypothesized that as the tree order increases, writes become slower because of increased node size, but reads become faster due to caching (Section 5.4.1).

Results

**DS** Figure 6.9 clearly shows that as the tree order increases, the index performance suffers when using workload A (Figure 6.9a). This is mainly due to the increasing size of data shipped across the network as the order increases. For example, an update operation on a tree of order 128 will cause a leaf node with potentially 256 values to be (de)serialized and shipped over the network, even though only one value was updated.

For workload B (Figure 6.9b) however, using an order of 64 seems to give the best performance among the orders used. This is probably because read operations can take advantage of cached nodes. It seems that with an order of 128, update latency starts to dominate. Furthermore, it looks like the optimum tree order for updates is not the same for reads.

![Figure 6.9: Varying the tree order for the DS index.](image)

**QS** For QS, similar to DS, increasing the order seems to affect performance negatively when using workload A, as can be seen in Figure 6.10a.

However, as opposed to DS, the same can be said about workload B (Figure 6.10b). This means that the cost of updates might be very high relative to reads, even when query processor and storage system are on the same machine. Contrary to DS therefore, both read and update operations perform worse with increasing tree order. The spike with tree order 128 could be due to various factors such as replication at the storage layer, sudden burst of network traffic or lock contention in the index.

**Synthesis** Figure 6.11 compares both DS’s and QS’s update performance using workload B. For smaller orders (<= 64), QS is the clear favorite, although the
curves grow closer and closer as the order increases. At order 128 and ignoring the spike, both architectures' update performance are strikingly similar. This once again underlines the fact that updates are expensive, especially with large nodes, regardless of whether they are shipped across the network as in DS or moved within a compute node as in QS, with replication being a main factor. Also note that because the cache size is determined by the number of nodes and not an absolute value such as in bytes, higher orders should have better read performance. For our benchmarks, this is true for DS until order 64. For QS, this could mean that traversing a deep tree of smaller order (i.e. made of smaller nodes) is very fast in comparison to fetching large nodes from the BDB store.
Since in this experiment, we do not change the cache size but adapt the initial population load, the trees of each order are of similar height. Using the default settings, the highest tree we construct has at most 7 levels; this is the case for DS when the order is 8 and we insert 1’600’000 values. QS’s upper limit is 5 levels, because we spread the nodes evenly across three compute nodes. This also means that higher order indexes can actually cache more data. It seems, however, that the performance impedance from fetching large nodes from the underlying store is not compensated by having more data in cache in the QS architecture, as opposed to the DS architecture. We look into why QS’s performance decreases as order increases in Section 6.2.2 and try to give bounds as to where the optimum tree order lies for both architectures.

Looking at Figure 6.12 which shows the latency-tree order plot given fixed maximum throughputs for reads of workload B, our hypothesis (Section 5.4.1) seems to hold, at least for DS. For QS, the plot looks like the second ‘half’ of the graph in our hypothesis, perhaps implying that smaller orders should be used.

Figure 6.12: Read latencies of DS and QS using workload B with increasing tree order using fixed maximum throughputs (in parenthesis).

### 6.1.3 Payload Size

#### Experiment

In this benchmark, we used the payload sizes 10, 100 and 1000 bytes. This means that using the default settings, a leaf node can contain up to 320, 3200 and 32’000 bytes of data respectively. As the payload size increases, performance should decrease in all architectures (Section 5.4.1).

#### Results

**DS** As expected, performance suffers as payload size increases. Looking at Figure 6.13a, it becomes clear that shipping 32’000B nodes across the network and (de)serializing such large nodes are costly. This again is most likely because updating incurs an additional network hop plus extra (de)serialization steps.

Notice that for workload B (Figure 6.13b), compared to its update performance, the read performance when using payload sizes 10B and 100B are similar, further highlighting high update costs.

**QS** Figure 6.14 shows similar general behavior of performance when increasing payload size. As with DS, latency hits the roof when using a payload of size
1000B. With workload B (Figure 6.14b), one can see that reads with payload sizes 10B and 100B do not perform very differently, but that for workload A (Figure 6.14a), using a payload size of 10B performs over 2.5x better. Once again, this could indicate that for an update operation in QS, the dominating factor slowing down performance is the (de)serialization and/or the underlying storage layer’s replication overhead.

**KII** Figure 6.15 depicts an interesting behavior of KII: There is almost no difference between using a payload size of 10B and 100B. This might be because Cloudy handles writes well for small payloads. For larger payloads (1000B), performance dips, which is expected as more data needs to be written to disk.

**Synthesis** Comparing all three architectures, it can be seen in Figure 6.16 that QS is the clear winner in terms of performance for small payloads (10B). For large payloads (1000B), KII clearly outmatches the other architectures. This is most probably due to the (de)serialization and replication of larger data (QS and DS nodes vs. KII values) and traversing of the index.

As expected and can be seen in Figure 6.17 using workload B’s read operation and fixed maximum throughputs as an example, performance decreases linearly as payload size increases for all three index architectures (Section 5.4.1). Although KII seems to have almost the same latencies even when using large...
Figure 6.15: Varying the payload sizes for the KII index.

Figure 6.16: Comparing the performance of the different architectures using payloads 10B and 1000B.
payloads, more experiments with other payload sizes should show a linear decrease in performance as well. We plan to benchmark KII using a payload size equal to the average size of the leaf nodes when using QS or DS in future to provide a fairer comparison.

![Figure 6.17: Read latencies of the three architectures using workload B with varying payload sizes using fixed maximum throughputs (in parenthesis).](image)

### 6.1.4 Cache Time-to-live

#### Experiment

In this benchmark, we set the cache time-to-live for the three architectures to 0s, 5s, 30s, and 60s. Setting a cache TTL to 0s effectively means no cache is used. We hypothesized that using no cache is detrimental to performance for all architectures, and that using a cache will positively influence performance at the cost of data freshness (Section 5.4.1).

#### Results

**DS**  
Figure 6.18 shows the effects on performance of varying the cache TTL in the Shared-Disk architecture.

![Figure 6.18: Varying the cache TTL for the DS index.](image)

(a) Workload A - updates  
(b) Workload B - reads

Figure 6.18: Varying the cache TTL for the DS index.

The effect of caching in both workloads on both update and read operations is remarkable, but varying the cache TTL beyond 30s seems to have little effect on the overall performance. Setting the cache TTL to 5s degrades performance
slightly in comparison to setting it at higher values because nodes have to be shipped across the network more often, but then the data is fresher.

**QS** The effect of caching on performance for the Shared-Nothing index architecture is very pronounced as well, as can be seen in Figure 6.19.

![Figure 6.19: Varying the cache TTL for the QS index.](image)

An interesting observation can be made here in workload A (Figure 6.19a): Setting the cache TTL at 60s seems to bring about worse performance than setting it at 5s or 30s, at least towards the ‘end’. As the reason for this is not very clear, this experiment should be repeated several times to see whether this was just an outlier; looking at the plot in general strongly suggests that it was one. The trend in general for using a cache (TTL > 0s) seems to have a one-time positive effect in QS. This is also visible in workload B. The two peaks for cache TTL = 30s in Figure 6.19b could have been caused by e.g. increased network traffic, lock contention or the JVM garbage collector.

**KII** As with the previous two architectures, Figure 6.20 shows the fact that using a cache has big performance benefits for this architecture. This experiment shows that whether making a single network hop (Federator to index client) or a retrieval plus two network hops (KVS to Federator to index client) makes a big difference in terms of performance.

![Figure 6.20: Varying the cache TTL for the KII index.](image)
Synthesis  Comparing the different architectures also shows some interesting results. Figure 6.21 shows workload B’s read and workload A’s update performance with the cache TTL set to 0 and 30 seconds. Notice that when no cache is used i.e. when cache TTL is set to 0 seconds, both QS and KII perform similarly (Figures 6.21a and 6.21b), especially when using workload B. Contrast this to Figures 6.21c and 6.21d and one can see that using a cache, QS has by far the better performance for workload B in comparison to the other two index architectures, but KII fares better for workload A. This could indicate that despite having the query processor and data store together on individual machines, the overhead of updating a (relatively large) node locally is higher than shipping individual (relatively small) values across the network as is done in KII, especially when considering replication. QS performing better for workload B could be due to the fact that QS (and DS) caches entire nodes, so getting values which lie on the same node does not require accessing the local data store, while in KII, every new individual value not in cache must be shipped across the network. Both KII and QS provide fresher data (Section 4.9) and yet perform better than DS.

As an aside, Figure 6.22 shows that using DS with a cache TTL set to 60 seconds has similar read performance (Subfigure (b)) to using QS without caching, but that for updates, even with caching enabled, DS is visibly outperformed by QS (Subfigure (a)).

Our hypothesis in Section 5.4.1 seems to be supported by this experiment, as is depicted in Figure 6.23 using workload B’s read latency and fixed maximum throughputs as an example.
6.1.5 Number of Tree Nodes

Experiment

For the DS architecture, we vary the number of tree nodes i.e. B-Link Tree Servers from 1 to 7 with a step of 2. Read performance should increase because load is spread across more servers, potentially increasing concurrency and each server has its own cache. Update performance should not change as there is still only one updater, except in the case of only 1 tree node, because then there will be more lock contention and it might get overloaded (Section 5.4.1). It should be mentioned that the amount of stale data increases proportionally to the number of tree nodes.

Results

Figure 6.24 shows that performance does indeed scale with increasing number of tree nodes, but not indefinitely.

The read performance using the read heavy workload B (Figure 6.24b) indicates that it could scale beyond 7 nodes due to having more total cache and the load being spread. As mentioned, this comes at the cost of having old data unless some sort of synchronization is used. Surprisingly, update performance in workload A does not fare as well when using 7 tree nodes (Figure 6.24a), having similar performance to using 3 tree nodes. One possible reason is that the increase in concurrent access to the underlying storage negatively influences
its overall performance. It could also be that Cloudy did not repartition the
data at the time of running the experiment or that it simply is an outlier.

Using the read latencies of workload B with increasing number of tree nodes
and a fixed maximum throughput as an example, Figure 6.25 shows that our
hypothesis in Section 5.4.1 holds.

Figure 6.25: Read latency of DS using workload B with increasing number of
tree nodes using a fixed maximum throughput (in parenthesis).

6.2 Other Benchmarks

6.2.1 Scans

Experiment

For this benchmark, we used the modified workload E on both the DS and QS
architectures and set the maximum scan length parameter to 100 and 1000. Using
the default uniform distribution for the scan length, this means the lengths
of the scans are in the ranges [1,100] and [1,1000] respectively. Table 6.1 shows
the minimum and maximum number of nodes required and data sizes given a
scan length. As the current scan implementation of KII is very inefficient, no
benchmarks have been made with it yet. With increasing scan lengths, perfor-
ance should drop in both QS and DS index architectures, as more nodes need
to be retrieved and the sizes of the return messages which need to be sent across
the network get bigger.
Table 6.1: Minimum and maximum data sizes and number of nodes derived from the scan lengths. The number in brackets indicate the maximum number of half-full nodes as opposed to full ones.

<table>
<thead>
<tr>
<th>Scan length</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. data size</td>
<td>100B</td>
<td>100B</td>
</tr>
<tr>
<td>Max. data size</td>
<td>10'000B</td>
<td>100'000B</td>
</tr>
<tr>
<td>Min. # nodes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. # nodes</td>
<td>4 (7)</td>
<td>32 (63)</td>
</tr>
</tbody>
</table>

Results

Figure 6.26 shows the anticipated results: Both DS and QS architectures perform much better when the scan lengths are small. As seen in Table 6.1, increasing the scan length by a factor of ten increases the maximum number of nodes by a factor of 8 to 9. For DS, this puts the brakes on for performance, especially if these nodes are not cached as a lot more nodes have to then cross the wire.

![Figure 6.26: DS & QS using scan lengths 100 and 1000.](image)

Notably, even QS’s performance suffers a lot, although, as can be seen in Figure 6.27, having the query processor locally does not necessitate tree nodes having to travel across the network and therefore, it outperforms DS by about half an order of magnitude.

![Figure 6.27: Comparing DS & QS using scan lengths 100 and 1000.](image)
6.2.2 Tree Order Revisited

Experiment

In the previous tree order benchmark, we observed that the order influences read and update operations differently in DS but not so in QS. Also, QS seemed to perform worse with increasing tree order. To investigate this phenomenon further, another order experiment was done. This experiment differs from the first one in that the initial population load is fixed at 1,000,000 inserts with a payload of 100B per insert and that the cache size i.e. number of nodes cacheable is adapted such that it can cache at most 10% of the total amount of data i.e. 10MB. For example, using an order of 8, a leaf node can at most have 16 entries which equals 1600B when using a payload of 100B. Dividing the 10MB by 1600B gives us 6250, the number we would configure the cache size to be for this order.

Results

DS In Figure 6.28, the update and read latencies of workloads A and B respectively are shown using a tree order of 1, 8, 16, 32 and 64 with a fixed maximum throughput. As hypothesized, there seems to be an optimum tree order, which in general is different depending on the operation. In this case, the optimum order for both operations lies at around 8. In contrast, the previous tree order benchmark showed the optimum order for reads to be at 64, while updates performed best 8. This difference is likely due to the different experiment setup. The tree heights in the previous experiment were roughly the same for each order and the cache size was kept constant whereas in this case, the cache size is adjusted and the number of inserts is fixed, meaning that higher order trees are shorter. Looking at the results, this indicates that cache size has a bigger influence on performance than tree height especially for read performance. Note that although not displayed in the figure, an initial test has shown that having a tree order of 4 performs worse than 8 for both operations.

Figure 6.28: Order Revisited - The influence of different tree orders on latencies of the DS architecture using fixed maximum throughputs (in parenthesis).

QS Figure 6.29 shows how update and read latencies, each using a fixed maximum throughput, of workloads A and B respectively develop as tree order increases. This time, the optimum lies at around tree order 4 for both operations. This could also explain why for the first tree order experiment, performance only seemed to decrease as the order increased for QS.
Synthesis A possible explanation for lower order (smaller than the optimum) trees having worse performance is that the tree height becomes exponentially larger given a fixed number of inserts for decreasing orders. This means more nodes have to be fetched from the store (or cache), thereby increasing index search latencies. Note however, that the ratio of the average tree height of a given order and number of inserts to the amount of cache space available actually increases with the order, so the cache eviction probability when traversing the tree is lower for lower orders. For higher orders (larger than the optimum), even if the nodes do not leave a compute node as in QS, the fact of having to fetch larger nodes from the storage system seems to cause a performance drop.

6.2.3 TPC-W

In this section, we present initial results from the TPC-W benchmark using the DS and QS index architectures. The TPC-W benchmark uses its own TpcwRequestWorker (Section 4.3), which directly forwards the response byte array from the store thereby pushing the deserialization step to the client. It also seems that performance heavily depends on the thread pool size of each Nio Server: Setting it to too small a number prevents the maximum potential throughput from being reached and setting it too large causes too much network traffic, resulting in congestion and thus preventing the peak throughput from being reached as well.

Experiment

The TPC-W benchmarks are setup similarly to the YCSB ones. After the necessary components are started, the load is increased over time (EBs are added) (Section 5.1.2). For DS, we vary the number of tree nodes from 1 to 4 with a step of 1 while having 3 storage servers. Additionally, we also benchmark DS using 5 and 7 storage servers with 4 tree nodes each. When using two or more tree nodes, one of them is dedicated to serve updates only (and some read operations which require the latest values for session consistency and monotonicity). For QS, we experiment with 1 and 2 Federators and 3 storage servers as well as 2 Federators with 4 storage servers. When using more than one Federator, one of them has the updater role, analogous to the one in DS.
Results

**DS**  In Figure 6.30, we can see that the number of WIPS achievable by using 1 and 2 tree nodes does not seem to differ too much. This means that using a dedicated *updater* tree node does not significantly boost performance and that the bottleneck is the tree node serving read requests.

![Figure 6.30: Achievable WIPS given a number of EBs from benchmarking the DS index architecture using TPC-W varying the number of tree nodes (TN) and storage servers (S). When more than one tree node is used, one of them is set to be a dedicated updater.](image)

A similar observation can be made when using 3 and 4 tree nodes with 3 servers. Although adding an additional tree node from 2 to 3 does improve performance, the KVS becomes the new bottleneck. Therefore, adding a fourth tree node does not notably change performance.

Using 4 tree nodes and 5 servers gives a notable boost in performance in comparison to using 3 servers because network traffic can be better distributed among 5 servers. The limit of scalability, however, seems to be at 7 servers (when using 4 tree nodes), as increased inter-node communication at the storage layer or the network bandwidth limit being reached starts to hamper performance.

**QS**  Figure 6.31 shows that using the QS index architecture, around 4000 WIPS can be reached using 2 Federators and 4 storage servers. Also, adding an additional Federator hardly improves performance, indicating that the bottleneck is the Federator serving read requests, whereas an additional server significantly boosts performance because of better load distribution. QS seems to be able to scale quite well and further experiments are needed to find its scalability limit.

**Synthesis**  We have also benchmarked other Cloud systems using TPC-W. Our results on Microsoft Azure [28, 27] and Amazon RDS [4] showed 2015 WIPS and 1959 WIPS respectively. For those benchmarks, a single database instance was used. However, it is unfair to directly compare the TPC-W results of this work.
with said results because of the following. Firstly, we do not use a web server layer for the sake of simplicity. We believe, however, that even if such a layer were used, despite incurring more processing overhead, it will not be the bottleneck of the system. Secondly, our SUT provides lower consistency guarantees than that of a traditional RDBMS, which was used in both previously mentioned benchmarks. Finally, in both the DS and QS experiments, the entire data fits in the cache of each B-Link Tree Server, naturally boosting performance. Future benchmarks will configure the systems such that more direct comparisons can be made.

The TPC-W benchmark results have shown that both QS and DS perform similarly. However, using more servers, QS will most likely perform better due to increased update and read concurrency, smaller trees, and smaller sized data being transferred over the network. Both architectures seem to be able to scale, and more experiments are needed to see where their limits lie. The current scalability bottleneck for both architectures seems to be the network, as both can achieve around 4000 WIPS.

6.3 Optimizations & Limitations

Due to time limitations, we could not benchmark all parameters mentioned in Section 5.1.1, such as the number of trees. Also, for DS and KII, elasticity benchmarks had to be left out, as there were some problems with Cloudy. These experiments are planned for the near future.

There are a number of areas which can be tweaked and fine-tuned. Cloudy and BDB both have many parameters which can be adjusted. Examples are the number of verb handler threads (we use 200) in Cloudy and the B+ tree fan-out (we use the default 4096) in BDB. Also, the thread pool size of each Nio Server is configurable. If Nio Clients use connection pooling (as is used by the TPC-W
benchmarks), the connection pool size can be tweaked as well. Database tuning
is an art in itself and is not the focus of this work. Although we have run
all YCSB experiments with 100 threads which has proved to provide sufficient
throughput, it would be interesting to use 200 or more threads. However, this
was not possible due to permission restrictions on the servers used. In general,
all benchmarks should be repeated several times in order to get averaged values
and to check for outliers.

We also encountered what we think is a Java bug: When starting the YCSB
benchmarks, we occasionally get a \texttt{sun.nio.ch.Util.atBugLevel} bug which
kills a thread. The effect can be seen when running YCSB in loading mode
because not all \texttt{recordcount} inserts will be made. When running YCSB in
transaction mode, this means that e.g. instead of the specified 100 threads, only
99 will run. Although it does not impact our experiments and results in any
major way, we hope that this annoyance will be resolved as soon as possible.

Using ordered or unordered inserts can influence how a B-Link tree is build,
but will not affect the clustering of nodes in the KVS. This is because both Data
Shipping and Query Shipping architectures use UUID version 1 (Section 3.1.2),
and the KII architecture uses its own key generator (Section 3.2.2). This means
that there is an inherent clustering at the node level. Also, in the ordered case,
as is used in QS, some index nodes can quickly become hotspots, as opposed to
the hashed case, where the keys are spread, especially when using a distribution
such as Zipfian or Latest. Note that there can be clustering at the KVS and the
node levels. This could lead to performance degradation or skews. It should be
noted that although there might be initial clustering at the KVS level, a good
load-balancer (such as the one Cloudy has) will repartition the key space if it
finds the distribution of data too imbalanced. Optimally therefore, warm-up
runs and tuning would have to be made before actual benchmarking. Using
different clustering schemes and benchmarking them could also be of interest.
Chapter 7

Conclusion

Although Cloud computing is getting more and more widespread, Cloud solution providers have provided simple interfaces for users and in particular, developers, thereby pushing the responsibility of, for example, providing strong consistency, transactions and index structures to the upper application layers.

This thesis has explored the trade-offs of different index architectures in the Cloud with a focus on performance. The YCSB experiments have shown that the Shared-Disk index architecture consistently performs worse than the other two. The advantages of the DS architecture is that its storage system, Cloudy, automatically provides load-balancing and dynamic repartitioning of data. Arbitrary keys can be added without reconfiguration. However, it seems that the KII index architecture, which also uses Cloudy, shows better performance. One of the main reasons to use an index is to be able to do range queries, so if automatic repartitioning and load-balancing as well as range query support are needed, then DS might be the right choice.

The Shared-Nothing index architecture has shown to be able to reach the highest performance especially when using smaller payloads. This use case shows that in spite of the additional overhead, using an index structure can perform better than directly using a KVS. If larger payloads need to be supported, using the KII architecture performs well because it only needs to update single values as opposed to entire nodes. However, using caches and tree nodes whose sizes are determined in bytes instead of by the number of nodes and tree order respectively for future benchmarks will allow a fairer comparison between the two architectures. One of the biggest drawbacks of the (current) Shared-Nothing architecture implementation is that there is no dynamic repartitioning, so it does not have the flexibility and ease-of-setup as the other two architectures. In other words, it is not elastic. In general, increased data transfer over the network as well as (de)serialization times seem to be the two main performance-hindering culprits while caching and increased concurrency bring the most performance benefits. Additionally, we have presented some initial results using the TPC-W benchmark, showing that using indexes in the Cloud has promising potential.

We hope to have laid a foundation for index architectures in the Cloud and that this initial impetus can be carried forward to drive more future fundamental research.

In the follow section, we describe some points to be considered in future
experiments and some aspects requiring more work and research.

7.1 Future Work

Benchmarking

Future benchmarks can use smaller steps for the target granularity instead of 1000, especially for experiments such as scans, which have lower performance. Points of interest such as where the optimum tree order lies can also be honed in by running the experiments with more fine-grained parameter steps to get more accurate results. All experiments use primary indexes without reference values. As such, a next step could be to benchmark secondary indexes and primary indexes using reference values. Also, the benchmarks should be repeated using the in-memory KVS so as to get an idea of how the index architectures perform using a close-to-optimal KVS. Also, monotonicity or atomicity could be implemented for all index architectures. An additional statistics tool such as [38] would complement the benchmarks, giving information such as network usage and open file handles, so as to help analyze results and pinpoint bottlenecks. As our SUT has extra (de)serialization steps and two network hops more because we use an internal index client which can lie on a different compute node and therefore uses message passing, future benchmarks could either directly communicate with the underlying layer i.e. bypass the index client or modify the underlying layer to directly pass the response data from the storage layer as a byte stream to the index/external client, thus pushing responsibility of serialization and aggregating data to a higher level but most probably improving performance. This has already been done with the TPC-W benchmarks.

We have shown benchmarking results using YCSB and TPC-W. These experiments all ran on a local cluster. A more realistic test and one which would enable benchmarking costs would be to run TPC-W on a (commercially available) Cloud such as Microsoft Azure or Amazon EC2.

Referential Integrity

The current implementation of secondary indexes ensure referential integrity in only one direction; when inserting a value into a secondary index or updating a secondary index value, an additional lookup is made in the primary index to make sure that the value to be inserted or updated to actually exists. The opposite direction, when a change in the primary index should be reflected in all its secondary indexes, is in the pipeline. Deletion does not currently ensure referential integrity and could be implemented as well.

Serialization

Serialization can still be optimized in different areas. In particular, a stronger distinction between primary and secondary indexes can be made which would allow saving bytes when serializing values. For example, in a primary index, since its value (a set) always contains exactly one element, the set size does not have to be serialized if this fact is known. It might be worth looking into Google’s Protocol Buffers [18], which seem to have optimized serialization of objects to a
large extent. This could speed up (de)serialization and reduce the data size sent over the network, thus killing (or at least injuring) two big birds with one stone.

As part of serialization, data could be compressed before being sent across the network. This would use more CPU (which does not seem to be a bottleneck) and fully utilize the network bandwidth (which does seem to be a bottleneck).

**Dynamic Repartitioning for the Shared-Nothing Architecture**

We already mentioned that one of the main disadvantages of the current QS index implementation is that the partitioning is static. Future research could be done in trying to find a hybrid solution. For example, the load-balancer and repartitioner of Cloudy could be integrated in the Lookup Server, thus gaining the ease-of-setup and flexibility of DS while potentially harnessing the performance advantage of the QS index architecture.

**Extended Range Query Support for KII**

Although the current KII index architecture implementation does support range queries, it is really slow as it retrieves all key-value pairs from Cloudy before filtering out those which are out of range. New algorithms could be developed to make this more efficient and to support various key types such as integer and string.

**Using Bytes for Tree Node and Cache Size**

Instead of using tree leaf and index orders for the B-Link tree, the tree can be modified such that the maximum size of each tree node is specified in bytes. This would mean that the index nodes of a tree each could contain a constant maximum amount of pointers i.e. URIs, given that they are of constant size. Leaf nodes could then contain arbitrarily large data up to the specified size. The node splitting algorithm in the B-Link tree implementation would have to be adapted.

Similarly, the cache can be modified such that its capacity is specified in bytes instead of by the number of nodes. This would allow easier and more flexible configurations, especially when each leaf node can be of arbitrary size up to a given maximum as previously described.
Bibliography


