System Architecture - Distributed B-Trees
Version 1

Michael Kussmaul
Department of Information Technology, University of Zürich
kussmaul@nix.ch
DRAFT-VERSION

2nd July 2004

Abstract

This paper will give a short high-level overview on a proposed system architecture on how a distributed b-tree like data-structure could be built. This document will be revised and updated depending on future work.

1 Introduction

When sorted access to data is needed in a centralized environment, b-trees (balanced trees) and their similar flavours are commonly used. We will now discuss, how such a data-structure can be scaled to a distributed environment, to provide the same functionality as a centralized b-tree. The method discussed here does extend the algorithms found on b-trees with additional levels to get the desired distribution - one can say, that the data-structure at the end is not a distributed b-tree, but solves the same problems and gives the same interface to key-value data:

- It allows access for exact match results (e.g. lookup of a single key)
- It allows range queries (e.g. lookup of a range of keys)
- It implements *set(key, value)*, *get(key)*, *getRange(startKey, endKey)*, *browse(startKey)* and similar interfaces to set and get data from the distributed data-structure.

2 Architecture

The proposed high-level architecture is designed to scale from small systems (enterprise level) to very large systems (Internet level). The main idea behind is to reuse as much as possible already available techniques and software, to lower the risk of building a complete new system. This also has the advantage, if new trends or new problem solving techniques come up, it should be possible to incorporate them into this system. Another point is, that for different requirements (e.g. smaller systems, larger systems) some components can be tuned for their specific environment, while not changing other parts. Such a framework is shown here (similar frameworks can be found in [5, 6])

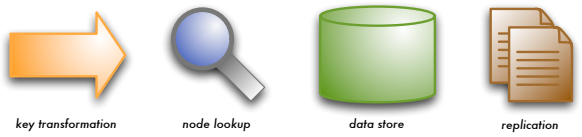

Figure 1: The different components for building a distributed data-structure

The next part of this article shows an overview how everything plays together,
and the following goes a bit more in detail of each component.

2.1 Overview

The main benefit of a distributed b-tree over a distributed hash-table is the sorted access to it's data.

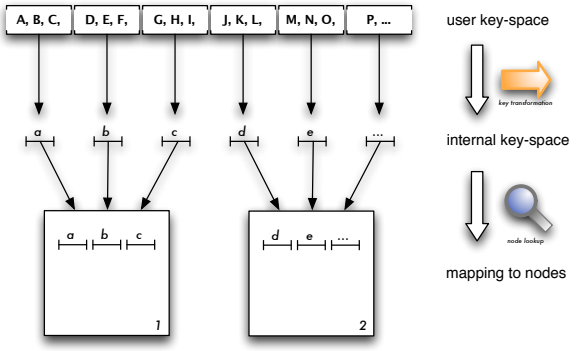

Figure 2: Mapping of user key-space → internal key-space → nodes

One idea would be to store the keys out-of-order (unstructured) and only during query-processing the order might be restored. But because the aspect of sorting is the fundamental piece of the whole system, it makes sense to sort the keys already at insertion time, so at all time the whole data-structure is always in a sorted state (structured).

Considering we have a user key-space $[A,Z]$, this range has to be partitioned and distributed among all participating nodes[1]. Distribution can be done on key-level, so each single key is managed by it's own (e.g. [4]), or we group some keys together into a group („*key-group*"), and we then manage the whole group as one single entity. A *key-group* is like a very small range of the whole key-space and all keys inside such a *key-group* $a \subset [A,Z]$ are ordered and are responsible for exactly this assigned range, no other *key-group* $b \subset [A,Z]$ contains a key which is also contained in *key-group* a (so $a \cap b \cap \ldots = \emptyset$). Each *key-group* has a unique label (id, hash-key) to identify it[2]. As an overview how the mapping of a user key to an internal key and the mapping to a node is done, take a look at Figure 2.

[1] At this point we don't look at data replication. So if one node goes away, all data this node was responsible are not accessible anymore - the data is lost.

[2] During this paper, elements of the user key-space are written in capital letters (e.g. A, B, C, ...) and *key-groups* are labeled with lowercase letters (e.g. a, b, c, ...)

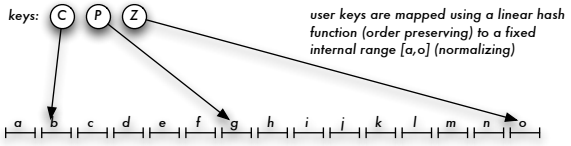

Figure 3: Transformation of keys to our internal key space

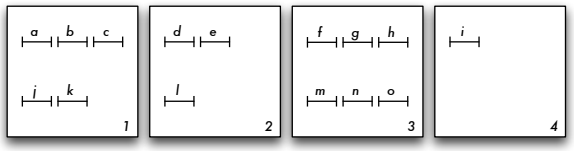

Figure 4: Range $[a,o]$ is distributed among 4 nodes

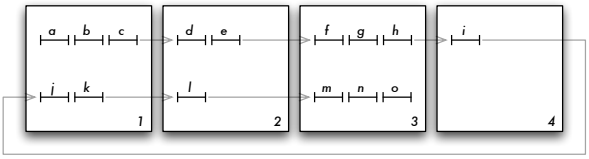

Figure 5: All subranges are linked to its successors and predecessors

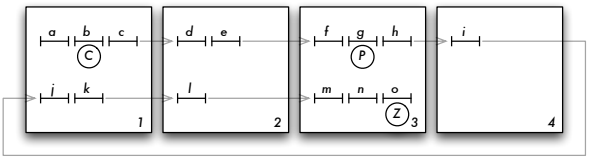

Figure 6: User keys C, P, Z are stored on the corresponding nodes

As the user of such a distributed data-structure wants to have the full flexibility over the keys - he might use alphanumeric, large range, small range, the keys can't just be used as received from the user. They must be transformed to an internal key range (example in Figure 3).

Assuming we have a whole internal key-range of $[a,o]$, then each participating nodes will be responsible for one (or more) non-overlapping subranges of the whole key-range (see Figure 4: e.g. node 1 consists of subranges $[a,c]$ and $[j,k]$) and all nodes know where the

successor and predecessor of their subranges is (Figure 5).

The transformation process depends on the size of the system. For example on a small 4 node system, it would be possible to have 4 large *key-groups* and just putting a 1:1 relation of *key-groups* and nodes, so each node is solely responsible for one single *key-group*. On a large system, there are much more *key-groups* and each only contains a very small subset of the whole key-range.

To retain the original user key, on the node itself the original key is stored (Figure 6)

The commonly used operations are described here:

2.1.1 Lookup / Storing of Key-Value Data

1. The user key is transformed to the internal key-space using a linear hash-function.

2. The corresponding node for this internal key is now searched using already proven P2P lookup methods (P2P overlays, distributed hash tables [16, 1, 11, 6, 7, 17]) or for smaller systems a simple lookup-table (static table, zeroconf implementation à la „Rendezvous" [2, 3, 9] or DNS like).

3. The corresponding node will receive the request and will return/store the data.

2.1.2 Lookup / Browsing a Range

1. The lower end of user key is transformed to the internal key-space using a linear hash-function.

2. The corresponding node is now looked-up.

3. The node will return all values in the requested range.

4. The node will forward the remaining request, which he can't fulfill to his next node (successor) and so on.

For optimization, range queries could also start from both ends, so the lower-end and the higher-end are executed in parallel, this will imply some kind of transaction-id, so if the two split processed will meet at one node (e.g. in the middle of the range) this node will know, that he already processed this query and will stop.

2.1.3 Node Insertion

1. The new node will send out a request to some nodes (local nodes) and asks them if they could migrate some *key-groups* to the new node. Perhaps a better approach can be found, so the nodes who are heavily overloaded will broadcast an S.O.S like token, so new nodes will better find overloaded nodes.

2. The new node will then register its successor and predecessor (This can be done in the way b-link trees register new tree nodes [10]).

3. And will then officially be the holder of the *key-group*.

2.1.4 Node Failure

1. If a node leaves the system (planned or unplanned) the links to its successor and predecessor will be broken.

2. It's successor and predecessor will search each other and will link together. The range of the lost node is not available anymore (except there was a data-replication mechanism in place), but the rest of the distributed data-structure is still functional.

2.2 Key Transformation

The key idea behind it is, that our implementation will have an already given key range for the internal key-space. Depending on the technology used for the node-lookup this can be a very large integer space. In case we would use „Bamboo" [13] as the underlying node-lookup infrastructure, the internal key space can be integers from 0 to 2^{160}. At the same time the user key-space could be phone numbers ranging from 1'000'000'000 to 9'999'999'999 and therefor need to be mapped to our internal key space. Also note that not the whole internal key-space needs to be used: In this example the phone number range could be mapped 1:1 into our internal key-space or better to an even smaller internal key-space, because otherwise each *key-group* would only contain a single entry[3].

This transformation needs to keep the ordering of the user key-space, that's why a linear hashing function

[3] E.g. the range 1'000'000'000 to 9'999'999'999 could be mapped to 0 to 1'000. This way each *key-group* would hold up to 9'000'000 numbers and we can still scale up to 1'000 nodes.

needs to be chosen. The hashing function does not need to provide perfect ordering, in case the hashing function only provides a proximity result (e.g. it found, that the requested key should be in *key-group a* but in fact is rather in *b*) the corresponding node which holds the *key-group* will just forwards the request to it's successor node, to query there. Of course this will have impact on performance, so the better the hashing function, the better the look-up performance[4].

2.3 Node Lookup

When we got the internal key for the request, we now have to look-up the corresponding host, which contains the requested key. All *key-groups* are distributed among all participating nodes. As a general rule, if we have several *key-groups* located at the same host, it is preferable that those *key-groups* will form a closed range of the internal key-space (e.g. it is preferred to have *key-groups* a, b, c, d on one node and not a, k, r, z, this will help preventing unnecessary jumps from one node to the other during range queries).

As mentioned above depending on the size of the system different approaches can be taken:

2.3.1 Smaller Systems

For smaller systems a static routing table can be put in place, e.g:

Internal Key	IP Address of Node
a	192.168.0.10
b	192.168.0.5
c	192.168.0.9
d	192.168.0.5

In a more dynamic environment, where constant nodes are added/removed DNS like approaches can be used or other local-network approaches like zeroconf (also known as „Rendezvous") [2, 3, 9] can be used, where every node broadcasts his internal key on the network, so others can easily find it. Zeroconf claims, that their broadcast mechanism is resource limited and will not overload the network with unnecessary broadcast messages. Never-less this mechanism does only work on the IP broadcast scope and is therefor limited to the a local network, as broadcast messages are not forwarded by a router.

2.3.2 Larger Systems

Obviously for a large scale system other mechanisms need to be considered, even non decentralized ones (E.g. for current Peer-to-Peer systems like [8]). In such systems the look-up becomes the most critical part of the system, as it is normally the slowest. Recently there are many projects working on a distributed look-up architecture (sometimes also called „Peer-to-Peer overlays" or „P2P overlays" for short) [16, 1, 11, 6, 5, 7, 17, 13]. Some of them could be used for our approach.

Depending on the underlying system we would chose, additional optimizations can be implemented. E.g. sometimes the the key space is not uniformly distributed, there exist some spots („data-spots") with lot of data and some with no data at all („data-holes"). In such a sparse occupied key range, not all *key-groups* needs to be pre-allocated. They could be allocated as soon as some data would fall into this *key-group*. This way during an insertion of a new key, the system asks the nearest node to also add this new *key-group* (E.g. if you already have *key-groups a* and *b* on node 1, and now a new key should be added, which would fall into *key-group c* the system will ask node 1, to also hold *key-group c*.) This approach has two advantages:

1. The *key-groups* do not need to be pre-allocated, this makes more efficient use of the nodes.

2. Allows better scalability, as newly added nodes can efficiently be filled with new keys, without having to migrate data from existing nodes.

To accomplish this feature, the underlying structure needs to support that look-ups for non-existing *key-groups* will be routed to the nearest existing *key-group* (e.g. if *key-group a* and f exist, and now a lookup for *key-group c* is done, the underlying structure needs to route this request to either *a* or f. This only has to be done for range-queries. For exact queries the system can immediately return, as the key does not exist).

2.4 Data Store

Each node has his own storage, where all the keys of this *key-groups* are stored. For efficiency reasons each

[4] This forwarding to successor node only needs to be done for range-queries. For exact queries the hashing function will never do any inaccuracy, because when the hashing function decides in which *key-group* to store the value, will be the same as when the user queries for the same value for a look-up.

node only needs to have one single storage, and all key-ranges can be stored on the same storage. As this storage also needs support for sorted access to keys, a conventional b-tree (e.g. Berkeley DB [15]) or highly concurrent b-link trees [12, 14, 10] can be used.

To make the system as flexible as possible and allowing further extensions, each key-value pair can have one or more additional meta-data information, which is stored alongside the data. Such meta-data information could be:

- Permission flags - so access to this data can be restricted to some users
- Time and date, transaction-id of last change
- How many times the data item has been accessed already
- Locks for transactions
- A flag if data is locally stored or stored on an other machine of the network
- etc.

This meta-data is also used by the system itself for monitoring, development and tuning purposes. Beside having meta-data and counters on data level, also other counters should be put in place on node level, e.g. the amount of network traffic coming in or out, how many queries the node already has answered, etc. This allows to monitor the whole system during testing, to find the best way to implement all it's features. Later it can also be used for self-maintenance, e.g. to detect a overloaded system, cache highly used data.

Each request coming to the node should have a „unique" transaction-id, so if the whole query is split in parallel and would reach a node twice - for whatever reason - the node will know, that he already processed this query and does not need to do any further action. Those transaction-id do not need to be stored for a long time, and can be removed after some time.

2.5 Replication

So far we have not discussed replication of data in such a system, as there are several aspects and granularity to look at, e.g. replication could be used as a active/standby mechanism, so the replication is only used in case the master is not available anymore, or in an active/active manner, where writing/reading could be done on either master or replica.

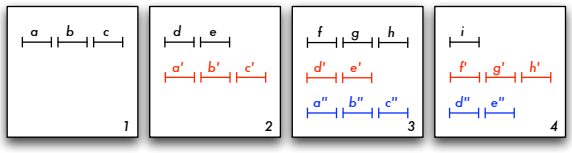

Figure 7: Replication: A master and two replicas

2.5.1 Active/Standby

This is the most simple approach and would only give the advantage of having a backup copy in case the master is not available anymore. Additionally several replicas could be used, to gain even higher redundancy in case of failure. This method would not give any performance advantage as having multiple data-sources for the same data, so access to it's data can't be load-shared among master and replicas.

But the advantage is, that this approach would not need to block write requests until all replicas could be updated, so this is rather simple to implement.

To include such mechanism into our distributed data-structure, a simple approach can be used: The original *key-group* acts as master and in case of an insertion/update of a key-value it will propagate this changes also to it's first replica, and this first replica will then propagate the changes along to the second replica (e.g. $a \rightarrow a' \rightarrow a''$). Alternatively, the master could also update all it's replicas (e.g $a \rightarrow a'$ and $a \rightarrow a''$). This could be done synchronously (the master waits sending back the response, until all replicas have been updated, and will then return it's status) or asynchronously (the master will immediately return it's status, and update it's replicas later).

The master and it's replica send always some tokens to probe their availability[5]. Alternatively replica 2 could also only probe replica 1, in case of a hierarchical setup.

If the master is not available anymore, the replicas will fail to send it's probe tokens and the next available will be elected as master. If we have an underlying node-lookup architecture as described in 2.3.2, the node-lookup will automatically find the correct replica, as he can't find the original host and will just forward the query to the next available *key-group* (e.g. if a

[5]Only the replicas have to send such a token, the master itself never needs to send a token to its replica, as he will know their status anyway, when an update fails.

lookup for *key-group a* fails, because node 1 is not reachable, the underlying node-lookup architecture tries to find the next available *key-group*. In Figure 7 this would be *key-group d* and this one is located on node 2 where also the first replica a' is located).

If the old master is available again, he will contact the current active replica to update it's value and will then act again as normal master.

2.5.2 Active/Active

With Active/Standby we gain redundancy in case of a node failure. With Active/Active we could additionally gain load-sharing, although this method is more difficult to implement.

A drawback of the current approach is shown, if some data which fall into the same *key-group* is „hot" and needs to be accessed from several clients at the same time. A potential bottleneck will occur, as only one single node is responsible for this data. Here replication could also help, in a way, that master and replicas allow access to it's data at the same time. Using this method synchronisation and locking between master and replica must be put in place, so all update requests must be atomic, and during an write/update, no client is allowed to access the same data for read/write. Depending on the transactional need, there are possibilities to tune the locking behavior, but never-less there is always some amount of administrative cost to update all replicas.

Generally we can assume, writing to data will be slower, as all replicas need to be updated and during that time no other client is allowed to access the same data. Each time before updating the data, the master has to set locks on each replica and can then update the values.

Reading from those data can then be in a load-balanced manner. The node-lookup mechanism will randomly (or any other load-balance mechanism can be used) forward the queries to either the master or his replicas.

Instead of using a separate key-space for all replicas, the node-lookup will just add a fixed prefix to it's internal *key-group*. This prefix has to be large, so the chance to put the replica on the same node as the master is minimized. Assume a prefix of 1000 and look at the master *key-group* with id „35". Doing a look-up of „35" will return node 1. As *key-group* „36" is probably also on node 1, we now take our prefix of 1000 and will then lookup *key-group* „1035" this one is most probably not on node 1 and on this node we could store our first replica, the second will be on the node with *key-group* „2035" and so on.

References

[1] The chord project. URL `http://www.pdos.lcs.mit.edu/chord`.

[2] Apple-Computer. Rendezvous, 2003. URL `http://www.apple.com/macosx/features/rendezvous`.

[3] Apple-Computer. Rendezvous technology brief, 2003. URL `http://images.apple.com/macosx/pdf/Panther_Rendezvous_TB_10232003.pdf`.

[4] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. URL `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2004-1926`.

[5] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. A storage and indexing framework for p2p systems. Technical report, Cornell University, 2004. URL `http://www.cs.cornell.edu/johannes/papers/2004/www2004-indexingArchitecturePoster.pdf`.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, pages 33–44, Berkeley, CA, Feb 2003. URL `http://www.cs.berkeley.edu/~ravenben/publications/abstracts/apis.html`.

[7] P. Druschel and A. Rowstron. Pastry - a substrate for peer-to-peer applications. URL `http://research.microsoft.com/~antr/Pastry`.

[8] J. Frankel. Gnutella file sharing. URL `http://www.gnutella.com`.

[9] IETF. Zero configuration networking (zeroconf), 1999. URL `http://www.zeroconf.org/`.

[10] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. Technical Report MIT/LCS/TR-530,

1992. URL http://citeseer.ist.psu.edu/johnson92distributed.html.

[11] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of dhts with openhash, a public dht service. URL http://iptps04.cs.ucsd.edu/papers/karp-openhash.pdf.

[12] P. L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. ISSN 0362-5915. URL http://www.it.iitb.ac.in/~it603/resources/papers/lehman.pdf.

[13] S. Rhea. The bamboo distributed hash table. URL http://bamboo-dht.org.

[14] Y. Sagiv. Concurrent operations on b-trees with overtaking. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 28–37. ACM Press, 1985. ISBN 0-89791-153-9.

[15] Sleepycat-Software. Berkeley db, 2004. URL http://www.sleepycat.com/products/db.shtml.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001. URL http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01.

[17] B. Y. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry - infrastructure for fault-resilient, decentralized location and routing. URL http://www.cs.berkeley.edu/~ravenben/tapestry.