Layering a DBMS on a DHT-Based Storage Engine

Eduardo A. Ribas[1], Roney Uba[1], Ana Paula Reinaldo[1],
Arion de Campos Jr.[1,2], Davi Arnaut[1], Carmem Hara[1]

[1] Universidade Federal do Paraná, Brazil
{ear07, rciu06, apd06, arioncj, davi, carmem}@inf.ufpr.br
[2] Universidade Estadual de Ponta Grossa, Brazil

Abstract. This article proposes an architecture for integrating a relational database management system (DBMS) with a distributed hash table (DHT). Systems developed based on this architecture provide functionalities of DHTs, such as scalability, decentralization and fault tolerance, combined with a high level query language provided by a DBMS. The main component of this architecture is the storage engine, responsible for implementing the interface between an SQL query processor and a DHT. We have implemented a system based on the MySQL DBMS and the Bamboo DHT, and conducted an experimental study to investigate two issues. The first determines the impact of developing an indexing structure for processing range queries. The second investigates two approaches for mapping relations to DHT's key-value pairs: vertical and horizontal partition of relations.

Categories and Subject Descriptors: H. Information Systems [**H.2. Database Management**]: Systems

Keywords: cloud database, DHT, P2P, storage engine

1. INTRODUCTION

Peer-to-peer (P2P) are distributed, non-hierarchical networks that have been proposed to provide resource sharing. One of the main properties of a P2P network is the absence of a centralized control over its structure. This property minimizes common problems found in centralized systems such as network congestion and waiting lists. For data management on P2P, distributed hash tables (DHT) are often used for storing data, while ensuring a number of desired properties, such as durability, fault tolerance, performance and scalability. DHTs store data as <`key, value`> pairs and provide a simple service interface composed of operations for storing a pair (`put(key, value)`), retrieving the value associated with a key (`get(key)`) and removing a pair (`rem(key)`). Given the similarity with disk-based operations, DHTs are largely used for sharing files, having file names as `keys`.

However, for applications that involve elaborate data manipulation, it may be a burden for the software developer to rely on a service that supports only data retrieval based on exact key match. These are applications that would ideally be supported by a database management system (DBMS), provided that the dataset is of manageable size and that there is little scalability concerns. In order to combine features of DBMSs and DHTs, in this article we propose an architecture for layering an SQL-based query processor on a P2P network. In the proposed architecture, the DBMS's SQL interface is maintained, providing high level operations for managing large volumes of data stored on a P2P network, while maintaining the complexities of data distribution transparent to the user. The main reason for using a DHT for storing data is to provide a massively distributed and scalable service that can potentially store huge amounts of data spread over thousands of machines. The integration of DBMS and DHT technologies involves many issues. One of them is an indexing structure to efficiently process range queries, that is, searches for data items with keys within a given interval.

This work has been partially supported by FINEP, project CT-INFRA/UFPR.

Systems based solely on DHTs are not able to process this type of query efficiently. This is because of the DHT standard interface that only provides exact key match searches. Another important issue is the mapping between the key-value pair model supported by DHTs and the relational model. In this article, we present some initial results from experiments that tackle these two problems.

The main contributions of this article are:

—Proposal of an architecture for integrating DHTs and relational DBMSs;

—Development of a system based on the proposed architecture, along with an indexing structure for processing range queries. Results from an experimental study are reported showing the impact of an index-based query processing;

—Analysis of two approaches for transforming relational data to key-value pairs. Results from an experimental study conducted to determine their impact on the system are also reported.

The rest of the article is organized as follows. Section 2 describes related work, while the integration architecture and our approaches for data mapping and range query processing are presented in Section 3. Our experiments involving these two issues are described in Section 4. We conclude in Section 5 presenting directions for future work.

2. RELATED WORK

Several systems and architectures have been proposed to support large volumes of data, such as cluster databases and noSQL systems. Cluster databases [Mattoso 2009] were proposed as high-scale relational repositories. These systems are composed of a cluster of commodity servers, each of them running a DBMS. On top of the set, a layer is developed for managing metadata, replication and query processing. The architecture proposed in this article differs from cluster databases by maintaining the same three layers approach of traditional DBMSs, but with a DHT service on the physical layer. While the goal of the majority of cluster databases is to execute OLAP queries in parallel, this article proposes an architecture for sharing and distributing data that are involved in simple and transactional processes. Alternative architectures for OLTP systems are evaluated in [Kossmann et al. 2010]. This study shows that systems, such as SimpleDB[1], MySQL Cluster[2][Ronström and Thalmann 2004] and MS Azure[3], achieve scalability through architectures based on replication, partitioning and caching. However, some systems present scalability limitations given that they rely on a centralized database server that maintains master copies. In addition, these systems do not present some important properties of traditional DBMSs, such as support for metadata and physical and logical independence. Our approach, on the other hand, maintains all the traditional features of a DBMS and combine them with a distributed and scalable data storage. [Curino et al. 2010] proposes a system with similar goals, but that follows a different architecture.

Efficient support for range queries is one of the challenges for integrating DBMS and DHT technologies. Many approaches have been proposed in the literature to tackle this problem. They can be divided into two categories: those that change the structure of the DHT and those that build an additional structure on top of a DHT. Mercury [Bharambe et al. 2004] is a system that follows the first approach. It changes the hash function and includes routing algorithms that support range queries and load balance. There are a number of previous work that follow the second approach, but here we mention two: PHT [Chawathe et al. 2005] and DST [Zheng et al. 2006]. A PHT is a search tree in which every node corresponds to a distinct prefix on the key domain. DST, on the other hand, is a segment tree.

[1]http://aws.amazon.com/simpledb/
[2]http://www.mysql.com/cluster/
[3]http://www.microsoft.com/windowsazure/

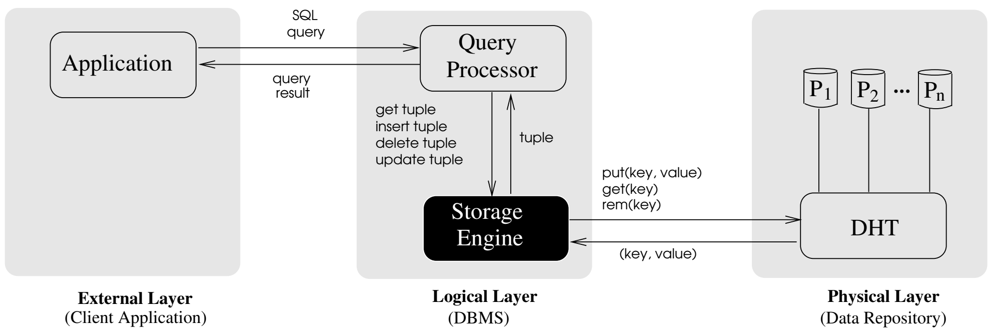


Fig. 1. DBMS and DHT integration architecture.

The distribution of data involves two important steps for determining the system performance: data fragmentation and distribution of the fragments. Data fragmentation can be on the level of relations, tuples or attributes. [Baião et al. 1998] emphasize that the type of fragmentation may impact the system performance. Besides horizontal and vertical partition of relations, their work also considers a hybrid approach for taking advantage of the best features in both. For the distribution of data, if no modifications on the underlying DHT are assumed, the node on which a given data item is stored is determined by the result of a hash function on its `key`. Usually, a hash function randomly distribute data among nodes that compose a P2P network to maintain load balance. However, the ability of controlling data location in DBMSs by clustering data based on keys has long been recognized as an efficient technique for speeding up query processing. In order to provide a similar functionality on DHTs, some datastores provide key-range partitioning, such as PNUTS [Cooper et al. 2008], App Engine's BigTable [Chang et al. 2006] and Spinnaker [Rao et al. 2011], or use a DHT-based system that maintains keys in lexicographical order, such as Scalaris [Schütt et al. 2008].

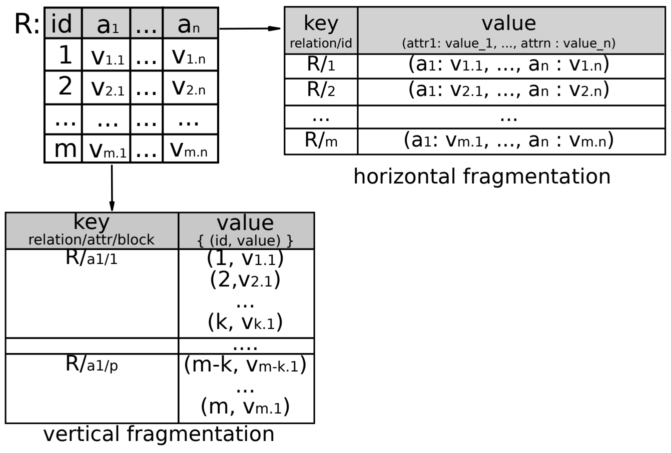
3. A DHT-BASED STORAGE ENGINE

The integration of DHT and DBMS technologies proposed in this article is based on the development of a storage engine. A storage engine is the underlying component of a DBMS, responsible for inserting, reading, updating and removing data items from physical storage. In a relational DBMS, this module receives requests expressed as operations on tuples and transforms them to operations on the storage device or service.

The proposed architecture is presented in Figure 1, showing the storage engine as the interface between the query processor of a DBMS and a data repository based on a DHT. This module is responsible for mapping data structured according to the logical model of the DBMS to a physical model based on key-value pairs supported by the data repository. It receives requests from the DBMS's query processor as relational operations on tuples and maps them to corresponding operations on the service interface of the data repository. This interface consists of standard DHT operations: `put(key, value)`, `get(key)` and `rem(key)`. One important property of the proposed architecture is that it is independent of the underlying DHT, given that it maintains the standard service interface. The DHT is responsible for the management and retrieval of data stored in the repository and provides a basis for relational query processing, while providing scalability.

3.1 Mapping Between Relational and Key-Value Based Models

To store relational data on a DHT we need to establish strategies for executing four basic tuple operations that the storage engine of a DBMS should provide: insertion, deletion, retrieval and update. As a design principle, we have implemented these operations using only the standard primitives exported by the DHT. For inserting a tuple into a relation, all attributes have to be packed in a key-value pair in order to be inserted in the DHT executing a `put` operation.


R:

id	a_1	...	a_n
1	$v_{1.1}$	...	$v_{1.n}$
2	$v_{2.1}$	...	$v_{2.n}$
...	...	...	...
m	$v_{m.1}$	...	$v_{m.n}$

key relation/id	value (attr1: value_1, ..., attrn : value_n)
R/1	(a_1: $v_{1.1}$, ..., a_n : $v_{1.n}$)
R/2	(a_1: $v_{2.1}$, ..., a_n : $v_{2.n}$)
...	...
R/m	(a_1: $v_{m.1}$, ..., a_n : $v_{m.n}$)

horizontal fragmentation

key relation/attr/block	value { (id, value) }
R/a1/1	(1, $v_{1.1}$) (2,$v_{2.1}$) ... (k, $v_{k.1}$)
	....
R/a1/p	(m-k, $v_{m-k.1}$) ... (m, $v_{m.1}$)

vertical fragmentation

Fig. 2. Data Fragmentation

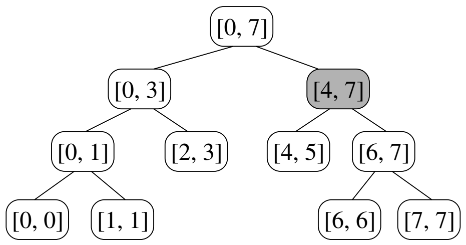


Fig. 3. Distributed Segment Tree

There are many ways of partitioning a relation to be stored as key-value pairs. The process of partitioning data is usually denoted as *fragmentation*. In the horizontal fragmentation, each fragment contains a tuple (or set of tuples) whereas in the vertical fragmentation each fragment contains values of the same attribute. Since each fragment is stored as a <`key, value`> pair in the data repository, we have to define how horizontal and vertical fragments are packed into a `value` and how they are identified by a `key`. We use the following approach. For the horizontal fragmentation, the `value` is a unidimensional array in which all attributes of the corresponding tuple is packed. A `key` is created by combining the name of the relation with the values of attributes that compose the primary key of the relation, in lexicographical order. As an example, consider a relation `R` with a primary key composed of a single attribute `id`, as illustrated in Figure 2. The `key` of a pair that corresponds to a tuple in which the value of `id` is 1 is "`R/1`" and the `value` contains all the attribute values. For the vertical fragmentation, the `value` is composed of a unidimensional array in which all the values of this attribute in the relation are packed. The key is generated by simply concatenating the name of the relation with the name of the attribute. Depending on the number of tuples of the relation, attribute values have to be packed into several blocks, instead of a single one. In this case, the `key` of a block must also contain the number of the block, as illustrated in Figure 2. That is, the pair containing the n-th block of attribute a_1 has `key` "`R/`a_1`/n`" and its `value` consists of a subset of attribute a_1's values. Observe that in order to be able to rebuild a tuple when the relation is vertically partitioned, each attribute value must be associated with its tuple identifier.

3.2 Indexing Structure

Once the mapping strategy for transforming relations to key-value pairs is defined, all relational operations have to be mapped to corresponding `put-get-rem` operations on the DHT. These operations provide a simple and efficient interface for retrieving individual data items. However, for retrieving all data items that contain keys in a given range, we need to execute a full scan over the relation. That is, since there is no way of retrieving only the proper subset of data, all tuples in the relation have to be retrieved for then selecting the ones within the interval. Among several data structures that have been proposed in the literature to support range queries on DHTs, the Distributed Segment Tree (DST) [Zheng et al. 2006] is defined based on the standard DHT interface, in accordance to our architecture principle. Moreover, it performs better than the PHT, which also follows the same approach.

A DST is a binary tree in which each node corresponds to a key interval (segment). The root node represents the full key range, while leaves contain a single key. Every internal node corresponds to a subrange of its parent and contains the union of all descendant intervals. An example of a DST is given in Figure 3. DSTs contain a user defined *saturation point* that defines the maximum number of keys a node may hold. In a tree in which nodes do not exceed this limit a key stored on a leaf is replicated in all its ancestor nodes. However, if a node reaches its limit, it is marked as *saturated.* A saturated node n that represents an interval $[s, t]$ does not contain all keys in the interval, warning the search algorithm that the search must continue on n's children to obtain additional keys. As an example, consider the DST of Figure 3. In order to retrieve all keys in the range $[4, 7]$, the search starts at the root node with range $[0, 7]$. The mid value of the range is 4, which determines that only the right child of the root has to be traversed. This node corresponds exactly to the search interval. If the node is not marked as saturated, then the search ends, since it contains all the keys in the interval. Otherwise, the search continues on both children in parallel and nodes corresponding to ranges [4, 5] and [6,7] are visited. If both nodes are not saturated the search ends returning the union of all keys stored on them. One important aspect of the DST is the fact that nodes are also distributed on the DHT. A pair <`key, value`> is generated for each node in which the `key` corresponds to the range `[s, t]` and the `value` contains the key set in the interval.

Next section describes the development of a prototype that follows the proposed architecture and discusses experimental results that show the impact of horizontal and vertical data fragmentation of data as well as of the addition of a DST indexing structure.

4. EXPERIMENTAL STUDY

For the development of a system based on the proposed architecture, we have chosen the MySQL DBMS and the Bamboo[4][Rhea et al. 2005] data repository. Bamboo is based on the Pastry [Rowstron and Druschel 2001] routing protocol and communicates using Sun RPC protocol on TCP or XML RPC on HTTP. In Bamboo, keys are limited to 40 bytes and its associated value cannot exceed 1024 bytes. The motivation for choosing MySQL is due to its decomposable architecture, with well-defined interface among its components. This simplifies the development of new modules, in particular, a new storage engine. The interface is based on a set of function prototypes that corresponds to functionalities expected from the storage engine. As an example, for inserting new tuples in a relation, the storage engine should implement a function `write_row(uchar* buf)`. It receives as input the tuple that is to be inserted in a compact form. Applying functions already provided by MySQL, it is possible to obtain the value of its attributes. Similarly, for making a full scan over a relation, MySQL defines two functions: `rnd_init()` to begin a new scan and `rnd_next(uchar *buf)` for retrieving tuples individually.

In the following sections, we describe two experiments that have been conducted to investigate two issues related to the integration of DBMS and DHT technologies: horizontal and vertical fragmentation of relations and range query processing.

4.1 Horizontal and Vertical Fragmentation

Both types of fragmentation described in Section 3, horizontal and blocked vertical fragmentation, were implemented. We have conducted an experiment using a relation with 2000 tuples, composed of 51 attributes of 20 bytes each. The total size of the relation is 2.04 MB. We have generated <`key, value`> pairs with `values` of 1024 bytes, which is the maximum size allowed by Bamboo. For the horizontal fragmentation, each tuple is mapped to a pair, with keys generated as described in the previous section. For the vertical fragmentation, each block contains 42 values of the same attribute.

[4]`http://bamboo-dht.org/`

The experiment was conducted on a local network with seven servers with dual or quad core processors and 6 to 32 GB of RAM, connected at 1 Gbps. Each server hosts ten nodes of the DHT and an additional gateway node was generated to receive messages sent by the MySQL DBMS; in total, the network was composed of 71 nodes.

The insertion of the entire relation using horizontal fragmentation took 1 minute and 39.87 seconds. For the vertical fragmentation, we applied blocked insertion. That is, the first 42 tuples were used to fill up 51 pairs that correspond to each of the relation's attributes and then the set of pairs were inserted in the DHT. This process took 1 minute and 59 seconds for the entire relation.

In order to compare the impact on data retrieval, we executed SQL queries to read all tuples in the relation, but varying the number of attributes in the result set. Figure 4(a) presents the experimental results. The vertical axis corresponds to query processing time, while the horizontal axis shows the number of attributes in the result. It can be observed that the query execution time when the relation is horizontally partitioned remains almost linear, not being affected by the increasing number of attributes in the result. This is because all tuples must be retrieved from the DHT, even when the final result of the relation is projected over a small number of attributes. For the vertical fragmentation, on the other hand, the execution time grows linearly. For retrieving the entire relation, it took 45.24 seconds for the horizontal fragmentation, while for the vertical partitioned representation, it executed in 1 minute and 29.50 seconds. This difference is due to three reasons. First, in the vertical fragmentation the volume of data is larger, given that each attribute value must be associated with the tuple's key. Second, the number of requests to the DHT is much larger for the vertical partition. In fact, in the experiment setting, with a relation of 2000 tuples and 51 attributes, divided in 42 values per block, the number of requests is $2000 \times 51/42 = 2.428$, while the horizontal fragmentation generates one request per tuple, that is, 2000 `get` operations. The vertical partitioning imposes roughly 21% overhead on the number of requests to the DHT. The third reason is the time spent for reconstructing a tuple, by joining all attribute values associated with the same tuple key.

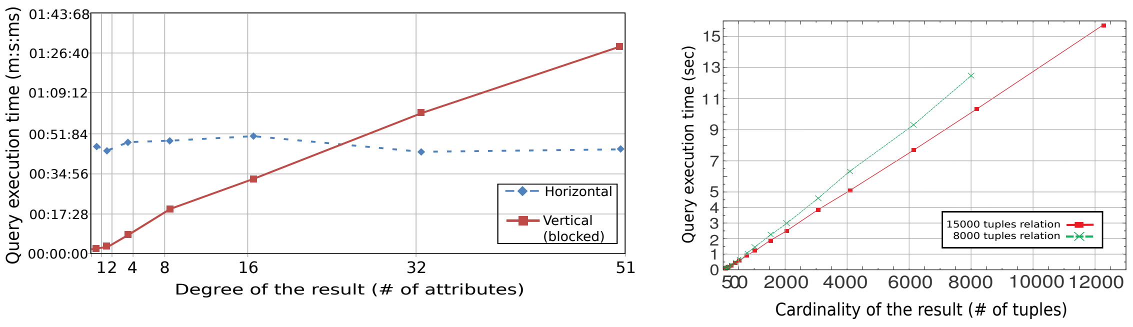
This experiment shows that vertical partitioning may be an interesting approach if the number of attributes involved in processing queries is small. In our particular setting, the processing times are the same when the number of attributes is half the number of attributes in the relation.

We have also conducted experiments with vertical blocks of smaller sizes. In the extreme case, when each block consists of only one attribute value, the query execution time was extremely high. For retrieving one single attribute, the query took 1 minute and 17.91 seconds, while 1 hour was spent for retrieving the entire relation. These results show that the number of requests to the DHT is a determining factor on the system's performance.

4.2 Range Queries

The goal of this experiment is determine the impact an indexing structure based on DST can have on the system. To this end, we have generated a DST index on the primary key of the relations. We have conducted experiments using two relations, r_1 and r_2, that follow the same schema, but with 8.000 and 15.000 tuples, and of size 1.1 e 1.5 MB, respectively. Both relations were stored following the horizontal fragmentation approach.

The experiment setting included ten servers with dual or quad core processors and 2GB or 4GB of RAM, interconnected in a 100 Mbps local network. Each server hosts five DHT nodes, composing a 50 nodes DHT. We have issued range queries on the primary key and the results of the experiment are shown in Figure 4(b). The vertical axis represents the query execution time and the horizontal axis, the number of tuples in the result. The reported execution time is the average of measures collected from the execution of 40 different queries that return exactly the same result size.



(a) Horizontal and Vertical Fragmentation

(b) Range Query

Fig. 4. Experimental results.

It can be observed that the response time grows linearly with the size of the result. The last marks of each plotted line show the time to retrieve the entire relation. For r_1 this takes 12 seconds and for r_2 the query runs in 16 seconds. Observe that without any indexing structure, these are roughly the time for processing any range query on these relations. This is because, without an index, the DHT does not provide any means to limit the search space. Thus, every tuple in the relation would have to be retrieved, delegating to the query processor the responsibility for filtering those of interest. This is equivalent to process range queries in a traditional DBMS using a full table scan. With a DST, in order to retrieve 2000 tuples from both relations, the system takes less than 3 seconds. Compared to the full scan on the DHT, these response times correspond to a speedup of 75% to retrieve 25% of r_1's tuples and 81% speedup to retrive 13.3% of r_2's tuples. One important observation is that by increasing the size of the result, the response times for r_1 grow faster than for r_2. This is because the DST's saturation point for r_1 was set smaller than for r_2. As a consequence, for retrieving the same number of tuples, more nodes have to be visited for r_1 than for r_2. Moreover, more nodes are likely to be saturated in r_1 and, as a consequence, descendant nodes have to be visited to retrieve all keys in the interval. This shows that the number of requests to the DHT is indeed a determining factor on the system's performance, confirming the results reported in Section 4.1.

5. CONCLUSION

This article proposes an architecture for integrating DBMS and DHT technologies. The main component of this architecture is the storage engine, which is responsible for mapping both data and operations defined on the relational model to key-value pairs and operations over a DHT. The architecture follows the three layers approach of a traditional DBMS, in which the DHT serves as the physical level component and the logical and external layers remain unchanged. One of the main advantages of this approach is that the distribution of data, provided by the DHT, is transparent to the user. The DHT also provides properties that are hard to achieve by a traditional DBMS, such as scalability, fault tolerance and decentralization. Moreover, it keeps all the expected functionality of a DBMS intact, such as support for metadata, logical and physical independence and support for a high-level query language.

Based on the proposed architecture, we have implemented a system for integrating the MySQL DBMS and the Bamboo DHT. One of the design principles in developing the system was the maintenance of the standard DHT interface, based on simple `put-get-rem` operations on key-value pairs. Data retrieval based solely on these operations requires full table scans to process queries that involve any operator other than equality on primary keys. As a first step towards solving this problem, we have implemented an index structure based on DST [Zheng et al. 2006]. Our experiments showed that for processing range queries, a DST can result in speedups of 75% to retrieve 25% of tuples in a relation, and 81% to retrieve 13.3% of the relation. Although the system allows joins and aggregations to be expressed as SQL queries, in the future we intend to investigate how these operations can be *efficiently* processed in our system.

We have also conducted an experiment to determine the impact that different forms of fragmentation have on query execution times. They show that the performance of the system is strongly dependent on the DHT and the number of requests sent to the DHT. Thus, in the future, we intend to investigate other issues related to database physical design, such as data clustering and secondary index structures. We believe that careful data fragmentation and distribution, combined with additional index structures, may have a huge impact on the system's performance.

Thus, the main contribution of this article is to show the feasibility of a system that integrates the scalability and robustness of a DHT with the ability to express complex queries in a high-level query language. Results from our experiments show the potentiality of the proposed approach and present directions for future work.

REFERENCES

Baião, F., Mattoso, M., and Zaverucha, G. Towards an inductive design of distributed object oriented databases. In *Proceedings of the IFCIS Conference on Cooperative Information Systems*. New York, NY, USA, pp. 88 – 197, 1998.

Bharambe, A. R., Agrawal, M., and Seshan, S. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Portland, OR, USA, pp. 353–366, 2004.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: a distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA, pp. 15–15, 2006.

Chawathe, Y., Ramabhadran, S., Ratnasamy, S., Lamarca, A., Shenker, S., and Hellerstein, J. A case study in building layered dht applications. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Philadelphia, PA, USA, pp. 97–108, 2005.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H. A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1 (2): 1277–1288, 2008.

Curino, C., Jones, E., Zhang, Y., Wu, E., and Madden, S. Relational cloud: The case for a database service. Tech. Rep. MIT-CSAIL-TR-2010-014, MIT. Mar., 2010.

Kossmann, D., Kraska, T., and Loesing, S. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Indianapolis, IN, USA, pp. 579–590, 2010.

Mattoso, M. Database clusters. In *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu (Eds.). Springer US, pp. 700–704, 2009.

Rao, J., Shekita, E. J., and Tata, S. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment* 4 (4): 243–254, Jan., 2011.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. Opendht: A public dht service and its uses. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Philadelphia, PA, USA, pp. 73–84, 2005.

Ronström, M. and Thalmann, L. Mysql cluster architecture overview - high availability features of mysql cluster. Tech. rep., MySQL Technical White Paper. Apr., 2004.

Rowstron, A. I. T. and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*. Heidelberg, Germany, pp. 329–350, 2001.

Schütt, T., Schintke, F., and Reinefeld, A. Scalaris: Reliable transactional p2p key-value store. In *Proceedings of the ACM SIGPLAN Workshop on ERLANG*. New York, NY, USA, pp. 41–48, 2008.

Zheng, C., Shen, G., Li, S., and Shenker, S. Distributed segment tree: Support of range query and cover query over dht. In *Proceedings of the International Workshop on Peer-to-Peer Systems*. Santa Barbara, CA, USA, 2006.