RIPPNET: Efficient Range Indexing in Peer-to-Peer Networks

Norvald H. Ryeng and Kjetil Nørvåg
Norwegian University of Science and Technology
Department of Computer and Information Science
Sem Sælands v. 7–9, 7491 Trondheim, Norway
{ryeng,noervaag}@idi.ntnu.no

Abstract

Write-heavy applications present a challenge to peer-to-peer indexing methods which need to update the index for each write operation. The costs incurred when the distributed index is updated becomes a bottleneck. Current distributed indexing methods are designed for indexing and retrieving single tuples, giving a very high update cost. In this paper we present a new approach to efficient peer-to-peer range indexing that employs indexing of ranges to reduce average update costs as well as providing efficient data localization and decoupling from data placement policies. Based on results from experiments, we demonstrate the applicability and significantly reduced update cost of the new approach.

1. Introduction

Large, distributed simulation processes can produce a lot of data. In some simulations the number of write operations are much higher than the number of read operations, and data is more often read from local storage than from other nodes. A distributed relational database management system (RDBMS) for large simulations, e.g., in a computational grid, must be able to efficiently perform writes, while at the same time allowing global lookups. The distributed, self-organizing nature of peer-to-peer systems provides a good basis for building such RDBMSs.

Existing peer-to-peer RDBMSs use tuple based indexing, which requires the distributed index to be updated every time a tuple is inserted, updated or deleted. For write-heavy systems, the cost of updating the distributed index is a bottleneck. In this paper we present RIPPNET, an indexing method for peer-to-peer RDBMSs that instead indexes ranges of tuples. The local database at each node is divided into ranges that are registered in the distributed index. By indexing ranges instead of tuples, the number of messages used to keep the index up to date is reduced, while still allowing for data localization queries.

The common indexing method for structured systems is to use a distributed hash table (DHT) that indexes tuples. Since the tuples are inserted into the DHT, the DHT enforces a certain data placement based on the hashing function. Our range index is orthogonal to any data placement policies. This is important in a simulation setting where each node mostly accesses the data produced locally, but occasionally needs to access data from other nodes. If the index structure dictates a data placement policy that is incompatible with the pattern produced by the simulation, the result is a lot of network traffic to write or read data. By decoupling indexing from data placement, many reads can be made local.

Thus the main contribution of this paper is an indexing method for peer-to-peer RDBMSs that:

- indexes ranges of tuples,
- significantly reduces costs for write-heavy systems, and
- is decoupled from data placement policies.

The described indexing method is implemented in a simulator and results from experiments show that update costs are significantly reduced.

In this paper we start by reviewing related work in Section 2 before we present some preliminaries in Section 3. Our new range indexing method is presented in Section 4, and extensions to the basic method are described in Section 5. Finally, we evaluate our approach and achieved results in Section 6 and conclude the paper in Section 7.

2. Related Work

Current systems typically use DHTs such as Kademlia [12], Chord [18], CAN [15], Pastry [16] or Tapestry [10] to store tuples or tuple indices. These systems support only exact match lookups. More advanced queries, such as range and cover queries, are blocked by the hashing function, which destroys the data ordering.

Two techniques for performing range queries in peer-to-peer systems are discussed in [21]. The first is simply to give the same hash key to values within a range, thereby reducing the number of different hash keys and DHT lookups needed. The other technique is to create a multicast group for each range.

Gupta et al. [8] present a range selection technique for DHTs based on locality sensitive hashing (LSH). The method locates data in $O(\log N)$ hops in a N-node network. However, the suggested methods only give approximate answers. HotRoD [14] uses a combination of locality-preserving hashing function and replication to support range queries in a DHT-based system. In [1] and [4] data are distributes contiguously into a DHT-like ring, but without using the hashing function, relying on load balancing algorithms to maintain fairness in case of data skew. GChord [23] utilizes Gray coding to support range queries. Consecutive values differ in only one bit, and this fact is used to forward queries through the Chord network.

Another approach is to organize the data into a tree structure. This class of systems include the Distributed Segment Tree [22], the Range Search Tree [7], BATON [11], P-Grid [2] and P-Tree [5]. A similar approach is search tries stored in DHTs [19].

The Distributed Segment Tree is a binary tree that can handle both range and cover queries. Unfortunately, nodes in all levels must be updated when a tuple is inserted or deleted. To reduce the load of higher-level nodes, these nodes can decide to drop tuples belonging to their children, but still the message has to be sent. In the Range Search Tree, each node of the tree consists of a load balanced set of physical nodes.

P-Tree is based on B^+-trees and uses Chord as the underlying DHT. Tuples are stored in leaf nodes, which constitute the Chord network. P-Grid places data in a binary prefix tree where each node maintains references to other nodes with the same prefix. Both P-Tree and P-Grid have worst case scenarios where the tree degenerates into a linked list. BATON overcomes this limitation by self-adjusting to data skew.

SkipIndex [20], SkipNet [9] and Skip Graphs [3] and ZNet [17] are based on Skip Lists, a tree of linked lists, where the list at level 0 is a linked list of all nodes. Higher level lists are increasingly sparse. Using these lists, range queries can be supported.

The RangeGuard system [13] uses a set of supernodes to allow for range queries in a DHT. The supernodes form a ring, each supernode taking responsibility for one range of the value space.

Common to all these range search strategies, are that they are based on tuple indexing. The cost of updates vary between the different structures, but all have to perform some work on every tuple update. Our approach is to store ranges in the index, thereby reducing the number of update messages sent to the index.

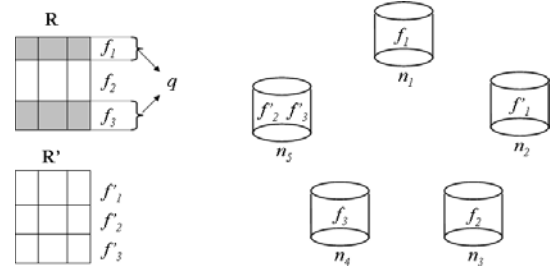

Figure 1. An example system.

3. Preliminaries

In this section we first present our system model and then the problem of distributed data localization.

3.1. System Model

In our model of a peer-to-peer DBMS, tuples are stored in horizontal fragments of relations. Relations are fragmented based on some fragmentation rule that may be local to a node or system-wide. A node may store more than one fragment of a relation, and these fragments need not be consecutive.

In the example shown in Figure 1, there are five nodes, $n_1, n_2, \dots n_5$. Nodes n_1, n_3 and n_4 each store a fragment, numbered f_1, f_2 and f_3, respectively, of relation $\mathbf{R} = \bigcup_{\forall i} f_i$. Node n_2 stores one fragment, f'_1, and node n_5 stores two fragments, f'_2 and f'_3, of relation $\mathbf{R}' = \bigcup_{\forall i} f'_i$.

It is our assumption that the data distribution is not uniform, at least not within a single node. Skewed data sets are common in real life applications, such as the distribution of names, and in our model we assume that this is generally the case.

For each fragment there exists a minimum and maximum allowable value for an attribute of a tuple in that fragment. Simple fragmentation rules may fragment based on only one attribute, e.g., separating the relation into fixed steps of the fragmentation attribute. This would limit the value of the fragmentation attribute in each fragment, but leave the other attributes limited only by the limits of the data type. The allowed range of the fragmentation attribute may be significantly larger than the range that is actually used.

3.2. Data Localization

The localization step in a distributed query processor needs to translate a query on global relations $\mathbf{R}$ and $\mathbf{R}'$ to a query on the physical fragments, $f_1, \dots, f_3$ and $f'_1, \dots, f'_3$, of the relations. Not all fragments are needed for all queries. If the query is for all tuples where an attribute is within some range, only fragments containing tuples within that range are needed. We call those fragments *relevant* to the query. Other

fragments of those queries are *irrelevant* to the query. In the example in Figure 1, query q separates the fragments of $\mathbf{R}$ into the relevant fragments, $\mathbf{R}_q = f_1 \cup f_3$, and the irrelevant fragments $\mathbf{R}_{\overline{q}} = f_2$. This concept of relevance is extended to nodes, such that relevant nodes contain at least one relevant fragment, and irrelevant nodes contain no relevant fragments. Given a query, q, we divide the set of nodes, $\mathbf{N}$, into those relevant to the query, $\mathbf{N}_q$, and those irrelevant to the query, $\mathbf{N}_{\overline{q}}$.

If the system is very small, it is more efficient to broadcast the query to all nodes, relevant or not, instead of first identifying relevant nodes and then send the query only to these nodes. The disadvantage of broadcasts increases with $\frac{|\mathbf{N}_{\overline{q}}|}{|\mathbf{N}_q|}$, the ratio of irrelevant nodes to relevant nodes. At some point, the cost of broadcasts exceeds the cost of identifying relevant nodes. Exactly when this occurs, depends on the cost of identifying relevant nodes.

When the system is very small, a complete index of all fragments can be stored on all nodes. As the system grows larger, this becomes infeasible. For each node to have complete knowledge of all fragments, even of all nodes, requires too much communication and state information. Currently most systems use DHTs, both to store tuples and to create distributed indices. The DHT allows all nodes to look up and retrieve data from all other nodes, with only a small amount of state information on each node.

One disadvantage of DHTs is that they only allow exact match lookups. If the relevant fragments for a query are all those within a range, we have to look up every single possible value within that range. The total cost of a query for a range using this method depends on the cost of a single exact match lookup and the width of the range in question. E.g., if there are only two possible values within the range, the two exact match queries required will not be a very costly range search. However, if the range is wide, the cost of looking up every possible value within the range makes it infeasible.

Efficient data localization is not without cost. For indices to be useful, they must be kept up-to-date. Maintaining an index of all tuples is costly. Every time a tuple is inserted in a relation in the system, a new index entry must be inserted. This detailed index is useful for some purposes, but overly detailed for data localization purposes. The data localization step only needs to find out which nodes are to be involved in the query, not the location of every single tuple.

A simple solution is to index ranges of tuples instead of single tuples. To find relevant nodes, the data localization step has to find all ranges that overlap the range requested by the query. For each range stored in the index, a node identifier is stored. A range query, q, will be answered by a set of relevant ranges and corresponding relevant nodes, $\mathbf{N}_q$.

4. Distributed Range Indexing

We propose a distributed index of ranges of tuples where each node defines ranges of tuples in its local database and stores information about these ranges in a distributed index. A query processor that needs to identify all nodes storing data within a range looks up in the index and finds all intersecting ranges.

The indexing method can be used on multidimensional data, e.g., indexing over multiple attributes of a relation. For ease of presentation, the examples will be limited to indices over one attribute.

To build, maintain and use a distributed range index, there are three main processes: range partitioning of data and building the index, maintaining the index, and searching the index. We will treat these processes separately.

4.1. Partitioning Local Data

Local data should be divided into ranges based on the indexing attributes. This could be done using using any clustering algorithm, or using fixed steps in the attribute domain. An incremental range partitioning method that allows for growing or shrinking of ranges is necessary. Using such a method, the algorithm would not have to rescan the whole database when a tuple is added.

Unlike many other applications, the ranges can be overlapping, i.e., a single tuple can belong to several ranges. However, index lookups will gain from having dense ranges, so adding tuples that are too distant from the rest of the range will probably not gain anything. Also, keeping the index updated when tuples are inserted, updated and deleted is easier when tuples belong to only one range.

Outliers may occur, and when deemed to far away for inclusion into one of the other ranges, they may form ranges of their own. However, since these outliers are in fact indexed as single tuples, the range partitioning algorithm should try to generate as few outliers as possible.

For each range identified, we create an index record. The only required fields of this record is the minimum and maximum values of the range and the network address of the node where this range is stored.

The index record can be further extended by storing various statistics on the range, such as the number of tuples, etc. This information is not necessary for the index to work, but may be useful to the query planner when constructing a query plan. This is discussed further in Section 5.3.

The data distribution on one node may look similar to that displayed in Figure 2. This node stores data that can be partitioned into four ranges, r_1, r_2, r_3 and r_4. These ranges cover only the parts of the attribute domain where the node has data. The ranges of one node need not cover all possible values. The example node has no outliers.

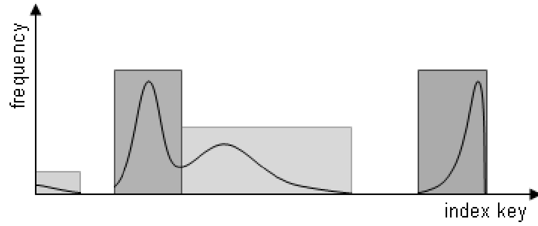

Figure 2. Data distribution on a node.

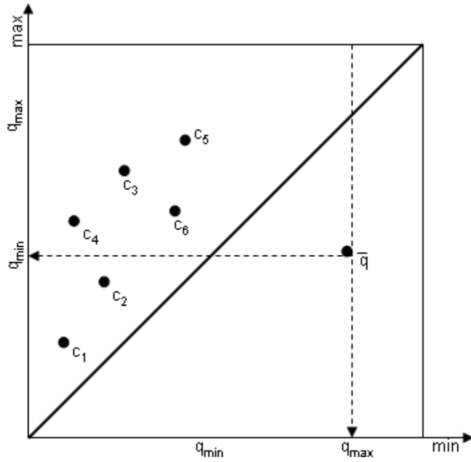

Figure 3. Two-dimensional index of one-dimensional data.

If the distribution is much more uniform than in our example, one may have to resort to fixed-width partitioning to define ranges. These will probably be sparser than what is outlined above, but the indexing method can still be used.

Each node has to identify ranges in its local database before the index can be built. Later, we will see how the ranges are maintained when tuples are inserted, updated and deleted.

4.2. Building the Index

When each node has partitioned its data into ranges, we can build a distributed index of all ranges. When indexing over d attributes, we build a $2d$-dimensional index. The dimensions are the minimum and maximum values of each of the d attributes.

Let us consider an index over a one-dimensional attribute, e.g., an integer value. Our index would then be a two-dimensional index, consisting of the pair of minimum and maximum values for all ranges. Each range, r, would be represented as a point, $\langle r_{min}, r_{max} \rangle$, in the index space, as shown in Figure 3.

Since the minimum value of a range is always smaller than the maximum value, the possible index records form a triangular space in the two-dimensional index. When partitioning data into ranges, some data points may be considered outliers and will be stored as single points in the index. Since the minimum and maximum values are equal for these points, they will be stored along the diagonal.

This structure can be stored in a Content Addressable Network (CAN) [15]. We have to bypass the hashing step and store the values directly in the CAN. The hashing algorithm makes sure data is evenly distributed among the nodes. When we bypass the hashing step, data will be unevenly distributed among the nodes. In particular, only the upper left triangle of a two-dimensional network will be used. In Section 5.2 we look at how we can keep a close to uniform data distribution in the network without the hashing step.

New index records are inserted into the index in the same way as for a normal CAN, except that the hashing step is bypassed. The index is updated when ranges in the original database changes. If a range is deleted or merged into another range, the corresponding index record is deleted. Updates and deletes are also similar to their normal CAN counterparts.

4.3. Maintaining the Index

Once the index has been created, it must be kept up-to-date. On inserts, updates and deletes, nodes must check if index records have to be updated.

On inserts, nodes must check if the tuples fit inside already existing ranges. If so, no further action needs to be taken. If not, it must decide if it should extend an existing range or define a new range to cover the tuples.

On deletes, nodes must check if it is possible to shrink the range the deleted tuples belong to. Too wide ranges does not affect the result, but it affects the performance of the index. Index records that describe a too wide range will result in more messages sent to nodes that do not store data within the requested range. If ranges are overlapping, there may be more than one range to check.

Updates are treated as a combination of inserts and deletes, and the corresponding actions must be taken.

To avoid having to look up in the distributed index for every insert, update and delete, nodes should store information about local ranges. This local information should be enough to decide when to extend or shrink existing ranges and when to create new.

If extra statistics are stored in the index records, care must be taken to also update this. Statistics updates may occur more often than ranges have to be extended or shrunk, so for this reason, exact count and similar statistics should be avoided.

The result of indexing ranges is that the index does not have to be updated for every insert, update and delete of

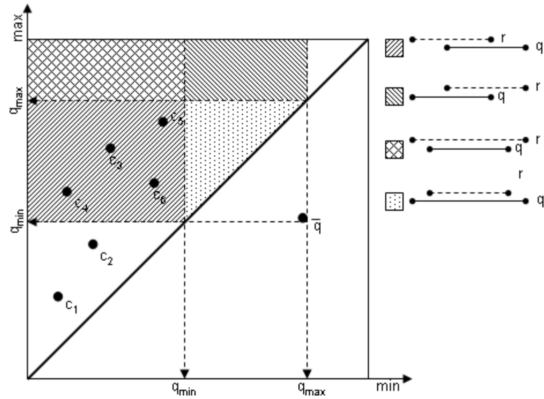

Figure 4. Range placement.

database tuples. The exact savings in communications costs depends on the data set and the frequency of such requests.

4.4. Index Lookups

The index allows for two types of lookups: range queries and exact match queries. Range queries are used to answer queries for indexed data, while exact match queries are used when updating index records.

4.4.1 Range Queries

There are four different situations that may occur when comparing two ranges. In our case, we will compare an indexed range, r, with the range of the query, q. The different situations that may occur are that

- r and q overlap in the lower part of r and the upper part of q,
- r and q overlap in the lower part of q and the upper part of r,
- q is contained in r, and
- r is contained in q.

The index is queried by ranges. A query q for the range $\langle q_{min}, q_{max} \rangle$ is represented in Figure 3 by its reverse range point, $\bar{q} = \langle q_{max}, q_{min} \rangle$, as shown in Figure 4. All points in the area delimited by the vertical axis, the line from $\bar{q}$ perpendicular to the vertical axis and the vertical line extending from $\bar{q}$ perpendicular to the horizontal axis represent ranges that overlap, contain or are contained by the range of q.

Query execution is done by routing a message to $\bar{q}$. The node containing $\bar{q}$ then forwards the query to its neighbors within the requested range, which again forwards the query to their neighbors, etc. In this way the query propagates through the network until it reaches all index nodes potentially containing overlapping ranges.

Any node that in this process receives the query, in addition to forwarding the query to its neighbors, responds to the querying node with a list of matching ranges.

As a result of this query propagation scheme, narrow ranges will be reached before wide ranges. These narrow ranges are also more likely to contain useful tuples. As shown in Figure 4, the first ranges that are encountered, are those that are contained within the query range, and thus guaranteed to only contain relevant data. The query then continues to partially overlapping ranges and ranges that contain the query range. The wider ranges are indexed on the last nodes to receive the query, and hence, returned last.

Cover queries are queries for regions that contain a certain point. The distributed range index can answer cover queries by answering the range query for ranges overlapping the range with minimum and maximum values equal to this point.

4.4.2 Exact Match Queries

The index can also be queried for an exact match, e.g., when a record needs to be updated. This is done in the same way as it is done in a normal CAN. This is a query type used to maintain the index. Exact match queries for indexed values are done as cover queries, since there may be more than one range covering the requested value.

5. Extensions

Section 4 presented the basic idea. In this section we present extensions to the basic idea for handling multidimensional data, load distribution and more advanced index records.

5.1. Multidimensional Data

The proposed range indexing technique supports multidimensional indices. Indexing d-dimensional data requires a $2d$-dimensional CAN, since for each dimension the index needs one minimum and one maximum dimension. The easiest mapping would then be to map minimum values to even numbered dimensions and maximum values to odd numbered dimensions, such that for each dimension i of the key, the index has two dimensions, $i_{min} = 2i$ and $i_{max} = 2i + 1$.

Since the dimensions are not independent, with increasing dimensions, a smaller percentage of the CAN is actually used to store data. The problem of uneven data and computational load distribution is discussed in the next section.

5.2. Uniform Distribution

An unfortunate effect of bypassing the hashing step of the CAN is that the data distribution is no longer uniform. For a two-dimensional CAN, half the address space is not used by the index. Also, the computational load is not distributed evenly. In Figure 3, the node storing the upper left corner of the CAN will be involved in every index lookup.

The naive approach to solving the data distribution problem is to swap maximum and minimum dimensions for different indices. If multiple indices are stored in the same CAN, they can use different dimensions as minimum and maximum dimensions, thereby evening out the data distribution. Also, the direction of dimensions can be switched. This does not guarantee uniform distribution, but helps in distributing data to all nodes, not only one half of them.

When dimensions are swapped and reversed, this also helps even out the differences in computational load, but still the corner nodes of a two-dimensional CAN will be more heavily loaded than other nodes. The most heavily loaded nodes could be replicated, using a round-robin algorithm to choose which replica to use for a specific index lookup.

5.3. Statistics in Index Records

Statistics on range size and distribution could be stored in the index record. If exact answers are not needed, this could be used to speed up aggregation queries, using the statistics to estimate the aggregate without asking the nodes where the data are stored.

The query planner can also use statistics to select first the nodes with a high density of tuples within the requested range. This could also be used to give a quick reply that gradually improves as the rest of the database is searched.

There are disadvantages to storing statistics in the index records. For the statistics to be correct, the index record must be updated more often than the range indexing mechanism requires. When adding statistical information to the index records, one must be careful not to require too frequent updates. The information in index records should change slowly and be defined as within some error margin, to avoid updating the index record for every single tuple insert, update or delete.

6. Experimental Evaluation

The proposed indexing technique was implemented in a CAN simulator that allowed us to experiment with different network sizes and database sizes. The proposed range indexing method is compared to a baseline method where queries are broadcast to all nodes, but only those nodes that contain relevant data reply.

6.1. Setup

The experiments were done using a Java-based CAN simulator extended with range query capabilities as described in Section 4. The simulator ran one query at a time, waiting until one query finished before the next was issued.

6.1.1 Network Model

For each of the network sizes, the network used was the exact same for each query. Nodes joining the network were given responsibility for zones as described in [15], using a random number to find the zone to split.

The simulated networks were static. There were no nodes joining or leaving the network during simulation.

Messages were forwarded through the CAN only when doing the actual lookup. The lookup message contained a node identifier of the querying node, and this identifier was used for direct communication outside the CAN. When overlapping ranges were found, the results were returned to the querying node directly using this identifier.

6.1.2 Data Set

A database was created for each node. The advantages of the indexing strategy is based on principle of data locality, i.e., data on a single node tends to be similar, or clustered. For each node a set of random seed points were chosen. From these seed points data clusters were grown.

An index was made over one attribute, a positive integer. The data on each node was partitioned into ranges using the DBSCAN [6] clustering method. Outliers were inserted as ranges consisting of single tuples.

6.1.3 Query Model

In each experiment, the network was queried 10,000 times. The range queried was chosen randomly, but the width of the query was fixed to a certain percentage of the attribute domain.

6.1.4 Metrics

For each query, we measured the number of messages used to propagate the query and return a complete answer. This number includes both the propagation of the query through the CAN and the direct return messages from nodes in the index to the querying node.

The first experiment describes the update frequency. In this experiment, the probability was measured in a network of 1,000 nodes, inserting the tuples in the order they were generated. For each node it was recorded whether the total range of the node had to be updated when a new tuple was inserted.

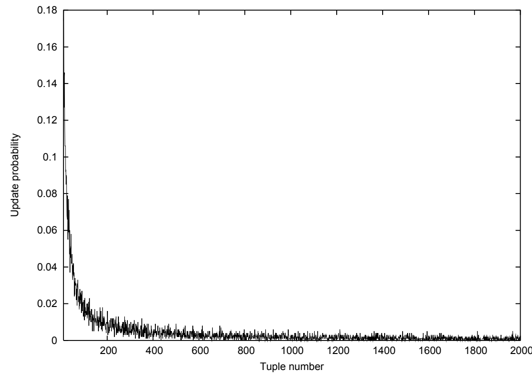

Figure 5. Probability of index updates.

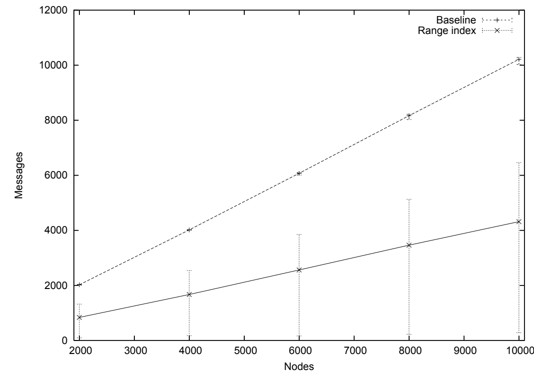

Figure 6. Messages used in varying network sizes.

6.2. Results

We now describe the results of three experiments. In the first experiment, we looked at the probability of index updates during tuple inserts. Then we looked at how the cost of querying varies with network size and query range.

6.2.1 Update Rate

Tuple indices must be updated on every tuple insert, update and delete. When indexing ranges, the update frequency can be reduced if new tuples that are arriving fall within a range that already is registered in the index. In this experiment we looked at the first 2,000 tuples inserted into each node of a 1,000 node network. The data set was the same as used in the other experiments, where points tend to cluster around a few points. Figure 5 shows the probability of tuples to extend the range and causing an index update.

The first 10 inserts have a very high probability of causing index updates, so they have been removed from the figure to allow us to see better what happens afterward. As we see from the figure, the probability of new tuples causing index updates is greatly reduced as the database fills up. Already after 10 inserts, the probability of causing an index update is reduced to 17.4%. After about 150 inserts, the probability of index updates is reduced to below 1%. This can be compared to the 100% probability of update in a tuple based index.

6.2.2 Varying Network Size

In this experiment we looked at the number of messages used to look up a range for varying networks sizes. The number of nodes in the network is the main parameter that decides the cost of a range lookup. The database used was a one-dimensional database of 10,000,000 tuples, distributed over networks of different sizes: 2,000; 4,000; 6,000; 8,000 and 10,000 nodes. The queries asked for a range covering 1% of the attribute domain.

From the results shown in Figure 6, we see that the average cost of lookups increase linearly with network size. The great variations in number of messages are a result of the area that is covered by a query. The query area is a function of the width of the query and the central point. Queries that are close to the edges of the attribute domain cover a smaller area of the CAN space than do queries covering the central part of the domain.

The number of messages used for range index lookup increases more slowly than the number used by the baseline, so the distance from the baseline increases with network size.

6.2.3 Varying Query Range

Our next experiment shows how the other important parameter, the width of the query range, affects the number of messages. The varying query ranges should also have an effect on the precision of the queries. The experiment was done on a network of 10,000 nodes, storing a database of 10,000,000 tuples. The number of messages used was measured for queries covering 1%, 20% and 40% of the attribute domain.

The results are shown in Figure 7. We see that the baseline method is nearly constant. The reason for this is that the only effect that increases the cost of this method is the number of ranges that are within the query range.

The cost of the range index method increases to more than the cost of the baseline method at a query width of about 37% of the domain width. After this point, the cost of locating nodes is higher than contacting all of them.

7. Conclusion

We have presented a method for distributed indexing of ranges instead of tuples in a peer-to-peer system. This indexing method significantly reduces the cost of inserts, updates

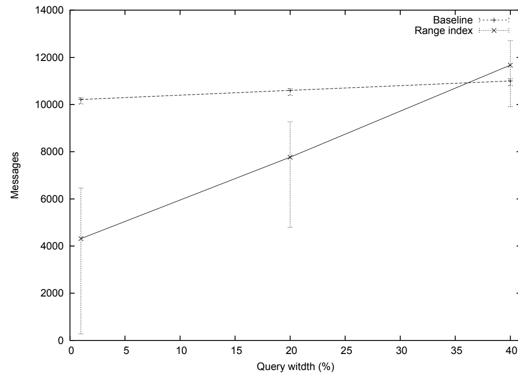

Figure 7. Messages used when varying query widths.

and deletes, since index records do not have to be updated every time a tuple is changed. The range index can be used for data localization in a RDBMS.

Unlike previous peer-to-peer indexing methods, data placement is decoupled from the index structure. This allows for a greater degree of node autonomy and greater flexibility in data placement.

We also look at the cost of index lookups and show that index lookups are more efficient for narrow searches, but that it at some point becomes more efficient just to broadcast the query to all nodes, since so many of them are involved anyway.

As far as we know, this is the first peer-to-peer range index, and there are still unsolved problems. Our index requires a specific kind of multidimensional DHT. Many systems use one-dimensional DHTs, and the method should be generalized to be used also in these systems.

More work should be done on answering queries by using statistics stored in index records. Summary queries, such as aggregation queries, will benefit from not having to check every tuple. Many systems require only approximate answers, and this would fit well in with the current indexing method, without requiring too frequent updates.

References

[1] M. Abdallah and H. C. Le. Scalable range query processing for large-scale distributed database applications. In *Proceedings of PDCS'2005*, 2005.

[2] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(3):29–33, 2003.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of SODA'2003*, 2003.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM'2004*, 2004.

[5] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of WebDB'04*, New York, NY, USA, 2004.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD'1996*, 1996.

[7] J. Gao and P. Steenkiste. Efficient support for range queries in DHT-based systems. Technical Report CMU-CS-03-215, Carnegie Mellon University, 2003.

[8] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of CIDR'2003*, 2003.

[9] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: a scalable overlay network with practical locality properties. In *Proceedings of USENIX Symposium on Internet Technologies and Systems'2003*, 2003.

[10] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of SPAA'2002*, 2002.

[11] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *Proceedings of VLDB'2005*, 2005.

[12] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS'2002*, 2002.

[13] N. Ntarmos, T. Pitoura, and P. Triantafillou. Range query optimization leveraging peer heterogenity in DHT data networks. In *Proceedings of DBISP2P'2005*, 2005.

[14] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Proceedings of EDBT'2006*, 2006.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM'01*, 2001.

[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware'2001*, 2001.

[17] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Proceedings of P2P'2005*, 2005.

[18] 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 SIGCOMM'01*, 2001.

[19] P. Yalagandula and J. C. Browne. Solving range queries in a distributed system. Technical Report TR-04-18, Department of Computer Sciences, University of Texas at Austin, 2004.

[20] C. Zhang, A. Krishnamurthy, and R. Y. Wang. SkipIndex: towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton University Computer Science Department, 2004.

[21] M. Zhang and K.-L. Tan. Supporting rich queries in DHT-based peer-to-peer systems. In *Proceedings of WETICE'2003*. IEEE Computer Society, 2003.

[22] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *Proceedings of IPTPS'2006*, 2006.

[23] M. Zhou, R. Zhang, W. Qian, and A. Zhou. GChord: indexing for multi-attribute query in P2P system with low maintenance cost. In *Proceedings of DASFAA'2007*, 2007.