

Distributed Data Summaries for Approximate Query Processing in PDMS

Katja Hose Daniel Klan Kai-Uwe Sattler
Department of Computer Science and Automation, TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany
{katja.hose|daniel.klan|kus}@tu-ilmenau.de

Abstract

Evolving from heterogeneous database systems one of the main problems in Peer Data Management Systems (PDMS) is distributed query processing. With the absence of global knowledge such strategies have to focus on routing the query efficiently to only those peers that are most likely to contribute to the final result. Using routing indexes is one possibility to achieve this. Since data may change over time these structures have to be updated and maintained which can be very expensive. In this paper, we present a novel kind of routing indexes that enables efficient query routing. Furthermore, we propose a threshold based update strategy that can help to reduce maintenance costs by far. We exemplify the benefit of these indexes using a distributed skyline strategy as an example. Finally, we show how relaxing exactness requirements, that are usually posed on results, can compensate the use of slightly outdated index information.

1 Introduction

Being a natural extension of federated database systems Peer Data Management Systems (PDMS) – also known as schema-based P2P systems – do not only inherit the benefits of such systems but also their disadvantages and problems. Especially, robustness and scalability do not come for free. Furthermore, in PDMS each peer might have an individual schema for its local data. Each schema might be unique in the whole system. Thus, PDMS have to support query rewriting and schema mappings. Regarding query processing it is not possible to contact an arbitrary peer in the system in a direct manner. Consequently, messages can only be sent to *neighbor* peers to which mappings are established.

Two of the main challenges for query processing in PDMS are (i) dealing with large-scaled settings and (ii) coping with dynamic behavior, autonomy, and heterogeneity. The first challenge demands to avoid flooding, since when we consider large numbers of participating peers asking each available peer is either much too expensive or even impossible. Characteristics of PDMS like peer autonomy and the avoidance of global knowledge mean further obstacles for query processing and especially for query routing: each peer has to decide independently from all the others

to which of its neighbors the query should be forwarded to. This decision has to be solely made on the basis of locally available knowledge. Such knowledge can be represented by *routing indexes* that describe the data that is available via each neighbor peer – typically not only the data of the neighbor itself but also that data that is available via the neighbor, possibly limited to a hop count horizon.

Since PDMS are dynamic systems data may change over time. In order to still route queries efficiently and correctly *indexes have to be adapted and maintained*. Unfortunately, this is a hard problem since propagating each single data update could paralyze the whole system. One possibility to counteract this problem of non up-to-date indexes is to *relax the exactness or correctness requirements* that are usually posed on results. This not only allows for reducing execution costs but also promises to compensate the influence of outdated index information to a certain extent.

In this paper, we identify a special kind of routing indexes (*Distributed Data Summaries*) that enable efficient query routing in PDMS. As an example we present the *QSummary* as a combination of tree-structures and histograms. To handle index maintenance we propose a *Threshold Propagation Strategy* (TPS) that effectively reduces the number of update messages. Finally, we address the problem of counteracting the influence of outdated index information by presenting a strategy for computing a *relaxed skyline* as an example for relaxed query operators.

2 Related Work

In *structured* P2P systems like CAN [11], Chord [13] or P-Grid [1] there is always some kind of global knowledge that tells us where to find the data we are looking for. In *unstructured* P2P systems – that form the basis of PDMS – peers are totally independent from each other. Each peer has its local data, its own schema and the ability to issue and answer queries.

Routing indexes are a crucial part of an efficient query routing. As defined in [4] they describe what data can be accessed by forwarding a query to a neighboring peer. More advanced indexes like those based on Bloom Filters [8] or histograms [9] have been developed. In principle, other data

structures that summarize data could also be used as the basis of routing indexes, see [2] for a survey. The problem that we encounter in PDMS is that we have to deal with structured data so that it is not sufficient to index data on instance level. We have to index data on schema level as well.

Another important problem is how to keep routing indexes up-to-date. Update propagation in P2P systems can be regarded as a replication problem where some peers have replicas of other peers' data. In that case replicas are not exact copies but summaries of the original data. A lot of techniques have been proposed for traditional replication problems in distributed environments [12]. Though the understanding of a replica is not the same for the problem at hand, the update strategy is a similar problem that has to be solved under the absence of any kind of global knowledge.

Traditional synchronous solutions for managing distributed data usually handle only a small number of peers and are not suitable for widely distributed environments like P2P systems where network connections are unreliable and peers are temporarily not available [6]. Epidemic algorithms [5] seem to be a good solution for distributed environments since they are robust against the failure of a single peer, do not make any assumptions on the network structure, and do not need any central instance for coordination. Just like in primary copy systems [6] each date in a PDMS belongs to one peer that is the only one that can update, change, or remove it. So we do not need to pay attention to conflict resolution strategies.

Taking these considerations into account, this paper (i) presents the *QSummary* as an example for routing indexes, (ii) proposes an efficient maintenance strategy, and (iii) shows how approximate query processing can reduce query execution costs in addition to applying routing indexes.

3 QSummaries - An Example for DDS

For an efficient query routing we need routing indexes that represent data, support query processing, and allow maintenance. More detailedly, they have to fulfill the following demands:

- summarizing data while restricting disk and memory space and minimizing the approximation error
- support of efficient look-ups
- caption of attribute correlations to efficiently process multidimensional data
- being easy to maintain and construct
- being all-purpose so that there exists no restriction on any specialized type of queries

We define *Distributed Data Summaries* (DDS) as that subclass of routing indexes that fulfills these demands. These let us focus on histograms and tree structures. Especially multidimensional histograms [10] fulfill the bigger part of

our requirements. But they have one major drawback: they do not cope well with sparse data. Tree structures and especially R-trees work well for describing multidimensional data. They support very efficient look-ups and can be constructed in an incremental manner. The only drawback of such traditional tree structures is that they neither approximate nor summarize data. As a result we combined these two approaches and retrieved the QSummary that we present in the following.

3.1 QSummaries

The data summary that each peer p maintains, has to summarize all the data that can be accessed by forwarding the query to p 's neighbors. To enable query routing the information about which data can be accessed via which neighbor has to be kept in a separate structure for each neighboring peer. Consequently, we need a base structure that summarizes the data of each neighbor.

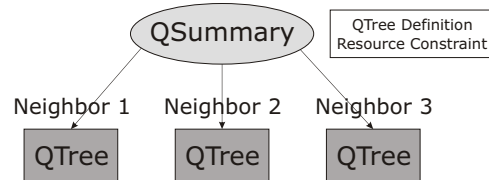


Figure 1. Example for a QSummary

Figure 1 illustrates the structure of a QSummary as an example for DDS. It maintains one QTree (base structure) for each neighbor. QTree parameters are defined at the QSummary level. Resources are restricted for all QTrees altogether, so if it reduces the overall approximation error, it is possible to use the major part of the resources to describe the data of just one neighbor.

3.2 QTree

As the QTree is a tree-like structure it consists of nodes each of which defined by means of a rectangular multidimensional bounding box. Such a minimal bounding box (MBB) describes the region in the data space that is represented by the subtree with the box owning node as root node. Thus, just like in R-trees, a child node's bounding box is always completely enclosed by the one of its parent. In order to limit memory and disk space we replace subtrees with special nodes: *buckets*. Only buckets contain statistical information about the data points that are contained in their MBB. The smallest buckets consist of only one point. Although buckets may contain almost any kind of statistical data, in this paper we will only consider buckets providing the number of data points. Each QTree is characterized by the parameters f_{max} and b_{max} . f_{max} describes the maximum number of child nodes (fanout) which each inner node may have. b_{max} represents the maximum number of buckets in a QTree – limiting resources.

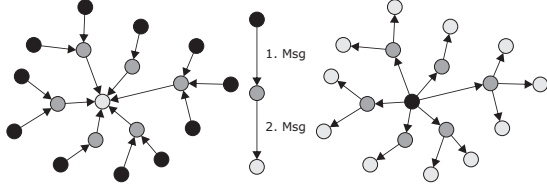


Figure 2. Construction

4 Construction and Maintenance

Concerning construction and maintenance there are three special aspects to consider:

1. constructing DDS for a fresh network
2. peers joining and leaving the network
3. peers updating their local data

The second aspect can be regarded as a special case of the third one. The main reason for this is that a peer that leaves or joins the network can send an update message to its neighbors. Such a message encodes that all the peer's data was removed or added so that the neighbors can adapt their DDS accordingly.

4.1 Construction of QSummaries

The construction of QSummaries can be regarded as flooding the network from the leaf nodes to the root (first phase) and back (second phase), see Figure 2 for an illustration. W.l.o.g. we assume the network graph to comply with a tree. For initializing the network we assume to know most of the peers that participate in the network. These peers could just as well be regarded as the super peers. Consequently, it is possible to synchronize the peers so that the first phase is started at the same time synchronously at the leaf nodes. These are all those nodes that have only one neighbor. All such peers send their local data by means of a QTree (in fact only its buckets) to their neighbors. These neighboring peers receive the data and add that information to their local QSummaries. If all neighbors except one have sent their QTrees then the receiving peer builds a QTree that contains the buckets of the received QTrees and its local data and sends the result to the one neighbor that has not yet sent its QTree.

In case data of all neighbors has been received, the second phase begins (Figure 2 right hand side). The peers send the received information along with information about its local data to its neighbors (for each message excluding the information originating from the receiving peer itself). Note that the possibility exists that there exist more than just one such peer. Once all leaf nodes have received the data the construction process is completed.

4.2 Maintenance of QSummaries

Whenever peers update their local data neighboring peers should be informed about that change so that they can

adapt their routing indexes accordingly. A well investigated asynchronous method for propagating information in distributed systems is epidemic algorithms [5]. Updates are immediately propagated to a randomly selected subset of the known peers. When the neighbors have received the updates and adapted their local data, they forward the updates in the same manner to a part of their own neighbors. Once started, epidemic dissemination is hard to stop. In order to prevent that updates are forwarded endlessly, a hop count or a time to life value can be used.

In case networks have cycles there are usually many paths on that information can reach a peer. Since in this paper we work with cycle-free networks, the information is not only sent to a subset of neighbors but to all of them, so that finally all peers are reached. In a network with cycles a peer may receive the same update multiple times. In order to still adapt its indexes correctly, a unique id is assigned to each single update. Consequently, peers are able to determine whether they have already received the update.

We propose a *threshold propagation strategy (TPS)* that propagates changes not until a predefined threshold of changes is reached. Before we present this strategy in detail we have to discuss two basic principles: (i) how to encode changes for propagation and (ii) how to measure changes of data. The first problem can be solved by using Add/Remove lists. Each entry is defined by the tag “add” or “remove”, the coordinates (i.e., indexed attribute values) of the updated data item, and a hop distance that indicates the distance of the peer which the change originates from. In case a hop count is defined this value allows for pruning the entries.

Solving the second problem means defining a measure for the changes in relation to the amount of data. We define the change of local data γ_{QL} as the average change of the corresponding QTree's buckets. The change δ of a bucket is twofold: size/enlargement and statistics.

The enlargement of a bucket's size is defined as follows:

$$\delta_1(b) = \frac{E_{new}(b) - E_{old}(b)}{E_{old}(b)} \quad (1)$$

where E denotes the maximum extension (length) of a bucket in all dimensions ($b.high$ and $b.low$ indicating bucket b 's upper and lower boundaries):

$$E := \max_{d \in \{1, \dots, d_{max}\}} \{b.high[d] - b.low[d]\} \quad (2)$$

As a measure for the change of a bucket's statistics we use the average percentage change of bucket b 's statistics, $stat$ denotes the number of data items that b is representing:

$$\delta_2(b) = \frac{\#changes\ in\ b}{\#data\ items} = \frac{|stat_{new}(b) - stat_{old}(b)|}{stat_{old}(b)} \quad (3)$$

In order to define an appropriate measure δ for the change of one bucket we have to combine the measures of Equations 1 and 3, obtaining:

$$\delta(b) = \omega_1 \cdot \delta_1(b) + \omega_2 \cdot \delta_2(b), \quad \omega_1, \omega_2 \in [0, 1], \quad \omega_1 + \omega_2 = 1 \quad (4)$$

where δ_1 and δ_2 are weighed according to the corresponding weights ω_1 and ω_2 ($\omega_1 = \omega_2 = 0.5$ as standard throughout

this paper). So the average change of all buckets that expresses the change that occurs in one QTree can be defined as follows:

$$\gamma_Q = \sum_{b \in B} \frac{\delta(b)}{|B|} \quad (5)$$

where B is the set of all buckets that the QTree contains.

A change propagation message received from neighbor i contains all the changes for that neighbor. So we can determine the change rate for neighbor i by adapting the QSummary's QTree for i and computing the corresponding change rate γ_{Q_i} as defined by Equation 5. Thus, we define the total change rate γ_T by the following equation:

$$\gamma_T = \frac{\sum_{n \in NB} \gamma_{Q_n} + \gamma_{Q_L}}{|NB| + 1} \quad (6)$$

where NB is the set of all neighbors.

Threshold Propagation Strategy. At first, the index structure (QSummary) is adapted, i.e., the QTree corresponding to the sender of the update message is adapted. In case local updates occurred the peer has already adapted the QTree for its local data in the same way. A QTree always remembers the updates that have not yet been forwarded to neighboring peers. If the total change rate γ_T exceeds the threshold τ then all known updates from the time when the last propagation took place up to this moment are forwarded to the neighbors. Note that we have to take care not to send updates back to the neighbor that they were received from in the first place. In case a hop count limit has been defined, we have to prune the updates accordingly. After all update messages have been sent the update records that are maintained by the QTrees are cleared.

4.3 Construction of QTrees

The QTree that we have already characterized in Section 3.2 is constructed by inserting one data item after another. In case there already exists a bucket B whose MBB encloses the data item, B 's statistics are adapted, otherwise a new bucket is created with only the data item as content. When inserting a bucket B we start at the root node and iteratively choose that child node whose MBB totally encloses B 's MBB. In case B is contained in more than one child's MBB we proceed with that child whose MBB center coordinates have the least distance to the center of B 's MBB. Formally, this is the child node n that minimizes c_r :

$$c_r(n, p) := \sum_{\substack{d \in \{1, \dots, d_{max}\} \\ n.low[d] \neq n.high[d]}} \frac{|\frac{1}{2}(n.low[d] + n.high[d]) - p[d]|}{n.high[d] - n.low[d]} \quad (7)$$

where $n.low[d]$ represents the lower bound of the bucket's MBB in dimension d , $n.high[d]$ the higher bound. In case several pairs have the same value for c_r we choose randomly between them. Having found node n_i that is the first to have no child whose MBB totally encloses B , B is inserted as a child of n_i . Of course, the constraints defined by f_{max} and b_{max} have to be checked and enforced.

Reducing the Fanout of Nodes. In case node n_i has too many child nodes after having inserted bucket B some of n_i 's child nodes have to be merged. At first, n_i determines that pair of child nodes (c_1, c_2) that minimizes the penalty function P_M :

$$P_M(c_1, c_2) := \max_{d \in \{1, \dots, d_{max}\}} \left\{ \begin{array}{l} \max \{c_1.high[d], c_2.high[d]\} - \\ \min \{c_1.low[d], c_2.low[d]\} \end{array} \right\} \quad (8)$$

Note that not only buckets are considered for merging but also inner nodes. We distinguish two situations: (i) $B \in \{c_1, c_2\}$ and (ii) $B \notin \{c_1, c_2\}$. In the former case if $c_{base} = \{c_1, c_2\} \setminus B$ is not a bucket, i.e., if it has further child nodes, B is detached and inserted as a child of c_{base} . If this results in a situation where c_{base} now has more than f_{max} children, the same actions have to be taken for c_{base} in order to merge some of its child nodes.

In case $B \notin \{c_1, c_2\}$ or c_{base} is a bucket, then a new *inter node* with c_1 and c_2 as children is created. In order to prevent the tree from degenerating into a list we try to drop the child nodes. A node can be dropped if all its children can be added to its parent without violating the fanout constraint.

Reducing the Number of Buckets. Whenever the number of existing buckets exceeds b_{max} some buckets have to be merged. So far we only consider to merge two sibling nodes at a time. In order to find that pair of buckets that causes the least penalty with low effort, we maintain a priority queue – each entry has a penalty value as key and a reference to a node as entry. In case node n that corresponds to the top entry of the priority queue has only two children, n is converted into a bucket – summarizing the data of both its child buckets. Note that only nodes with at least two child buckets have an entry in the priority queue. In any other case, those two bucket child nodes (c_1, c_2) are merged whose penalty is the least. The resulting bucket finally replaces c_1 and c_2 . Since the node has now less children than before, we try to drop it.

5 Approximate Query Processing Using QSummaries

In distributed and dynamic environments without global knowledge we cannot guarantee the correctness of a result as we can do in centralized systems. The problem is that in large-scaled dynamic networks it is nearly impossible – or at least much too expensive – to ask all the peers in the system. Consequently, we have to relax the correctness/completeness requirements and find a routing strategy that enables efficient routing. The relaxation must somehow be quantified and output to the user as a result guarantee. In addition to the relaxation of completeness an orthogonal approach of reducing execution costs is to deploy approximating query operators.

In the following, we present a solution for the problems mentioned above: at first we point out how to use QSummaries for efficiently routing queries. Then we present a

concrete strategy that (i) uses DDS for routing queries efficiently, that (ii) deploys the skyline operator as an example for approximating query operators, and that (iii) gives distance-based guarantees for the result quality.

5.1 Routing Queries Using QSummaries

As already described in Section 3 each peer maintains one QSummary and thus one QTree for each neighboring peer. Figure 3(a) depicts a QSummary that peer p might possess. Assume p issues a selection query asking for data items with the coordinates indicated by the cross in Figure 3(a). As the illustration shows there exists no such data item. Having QSummaries we do not have to forward the query at all since there is no peer that might have an exact answer to the query.

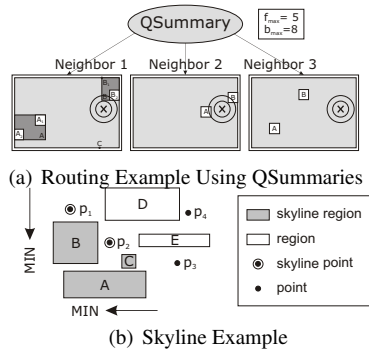


Figure 3. Integration of DDS Information into Query Processing

If the user does not require an exact answer, one could use the nearest neighbor selection as a ranking-based variant of the select operator. Given a distance (ranking) function d and a maximum distance ε we are also looking for data items within the region defined by d and ε . If p issues such a query and takes its QSummary into account, it forwards the query only to neighbors 1 and 2. This way, we can prune some of the routing paths that do not lead to any relevant results.

5.2 Skylines using DDS

In this section we want to present a strategy that additionally uses approximating rank-aware operators giving qualities for the result. For this purpose, we chose the skyline operator [3] that is formally defined as follows: For a given set of data objects a skyline consists of all those objects that are not *dominated* by another one. One object dominates another if it is *as good as or better* in all dimensions (attribute) and *better* in at least one dimension. This means when a user is for example looking for hotels (data items) that are both close to the beach (dimension 1: distance to beach) and as cheap as possible (dimension 2: price), the skyline contains all hotels that represent an “attractive” combination of minimizing both dimensions.

Skyline Over Regions. For processing skylines we have to consider not only local data items for evaluating the dominance relation \succ but also DDS regions. For this purpose, we have to generalize \succ to \succ_{Reg} that can be applied on regions A and B as well – regarding local data items as regions without extension:

$$A \succ_{Reg} B \Leftrightarrow a \succ b \quad \forall a \in A, \forall b \in B \quad (9)$$

\succ_{Reg} can be determined by means of a worst and a best data item. If an $a_{worst} \in A$ and a $b_{best} \in B$ can be constructed such that:

$$\forall a' \in A : a' \succ a_{worst} \text{ and } \forall b' \in B : b_{best} \succ b' \quad (10)$$

holds, then $A \succ_{Reg} B$ if and only if $a_{worst} \succ b_{best}$.

Figure 3(b) shows an example for a skyline over regions. In case both dimensions are to be minimized points p_1 and p_2 are members of the result skyline and regions A , B , and C provide further result elements. This is because only the following dominations occur: $A \succ p_3$, $A \succ p_4$, $C \succ E$, $C \succ p_4$, $p_2 \succ p_4$, $B \succ D$, and $B \succ p_4$. Note that $A \not\succeq p_2$ because a point in the bottom right corner of A would not dominate p_2 . Furthermore, $A \not\succeq C$ since it might be possible that data items exist in A that are located to the “right” of C and thus not all data items in A would dominate all data items in C . Considering this dominates relation \succ_{Reg} over regions only those neighbors providing information about relevant regions have to be asked in order to answer the query correctly.

Relaxed Skylines. The next step in reducing execution costs is to relax the correctness requirements that are usually posed on queries. Figure 4 illustrates the principle of a *relaxed skyline* using a two-dimensional data set. Assuming a skyline query asks for minimizing both dimensions 4(a) shows the exact answer with solid black circles indicating result points. 4(b) shows the corresponding relaxed skyline where many skyline points can be represented by only a few points (solid black circles). Each grey area illustrates the region that is represented by the one *representative* in its center – w.l.o.g. assuming Euclidean distance.

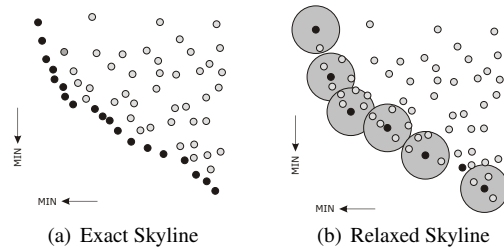


Figure 4. Illustration of a Relaxed Skyline

It is obvious that whenever a DDS region R fits into such a light grey area and the representative r is a local data item then R can be represented by r and the corresponding peer does not have to be asked for the data items described by R . Given a distance (ranking) function d and a maximum distance ε the algorithm for distributed processing of relaxed skylines works as follows:

1. Compute the skyline over the local data, DDS regions and points received along with the query.
2. Find representatives for all regions that remained in the skyline.
3. Forward the query to all peers that correspond to regions that could not be represented.
4. Having received the answer of all asked peers, determine the skyline over the local data, answer data and representatives. Since each representative defines a region the same principles as for the skyline over regions apply to the skyline over representatives.
5. Minimize the number of representatives. This is possible since one representative can represent more than just one skyline region.

Note that if we set ε to 0 the strategy computes an exact skyline – not using any representatives. As long as the index information is correct this strategy does not commit any error since relaxations are described by the guarantee of ε .

6 Evaluation

For all tests in this section we used three different setups. Each setup is based on the same cycle-free topology of 100 peers where each peer has 50 data items, has on average two to three neighbors, uses a maximum fanout (f_{max}) of 4 and a maximum number of buckets (b_{max}) of 50 for its QSummary. The characteristics that distinguish these setups are:

1. **Random Data, Random Distribution:** The attribute values of each data item and for all dimensions are chosen randomly (distributed uniformly) but restricted to the interval $[0, 1000]$. Each attribute value is determined independently from the others.
2. **Clustered Data, Clustered Distribution:** A cluster center is chosen for each of the 100 peers in the manner of random data. Each data item of a peer is determined by adding a random offset of $[-dist, +dist]$ (default $dist = 10$) to the cluster center’s coordinates.
3. **Anti-correlated Data, Random Distribution:** For each data item a point is chosen on the straight line through the points $p_1(1000, 0)$ and $p_2(0, 1000)$ and offset by $[-dist, +dist]$ (default $dist = 10$).

6.1 QTree

At first, we want to illustrate what factors influence the approximation quality of QTrees and thus QSummaries. These are: (i) the number of buckets b_{max} , (ii) the number of data items, (iii) the maximum fanout f_{max} , (iv) characteristics of the data distribution, and (v) the dimensionality, i.e., the number of indexed attribute values. The approximation error E is defined by Equation 1 – normalized by

the maximum possible extension of a bucket ($[0, 1000]$ for our data).

Number of Buckets. The most obvious influence has b_{max} : the higher the number of buckets the higher must be the approximation quality because raising the number of buckets allows the QTree to consume more resources. This effect is shown by all our QTree tests in Figure 5: the approximation error drops with higher b_{max} which means that the approximation quality rises.

Number of Data Items. Figure 5(a) shows the influence of the number of data items on the approximation quality using the random data set: the higher the number of data items the higher the approximation error. This effect is not shown by the anti-correlated and clustered data sets. The reason for this is that the other data sets have a predefined number of clusters that have to be summarized by buckets. The concrete number of data points in these clusters do not affect the approximation error in any notable manner. For random data, however, where there are no such clusters each new data item might increase the size of a bucket’s MBB and thus the approximation error.

Maximum Fanout. At first, we expected that f_{max} should not have any influence on the approximation error. But as we can see in Figure 5(b) it has. The explanation is that we only consider two sibling nodes for merging at a time. We do not check if merging non-sibling buckets would result in a lower penalty. So a higher fanout provides a better chance of finding a sibling bucket which results in a lower approximation error.

Data Distribution. As Figure 5(c) shows, the data distribution has a strong influence on the approximation quality. Random data has the highest approximation error because there are no clusters that could be summarized with a low approximation error. Consequently, we also see that the “bigger” the clusters (high $dist$) the higher is the approximation error.

Dimensionality. Figure 5(d) illustrates our results with varying dimensions using the random data setup: the more dimensions the higher the approximation error. The reason for this is the increase of the buckets’ sizes with a higher number of dimensions.

6.2 TPS

Figure 5(e,f) show our results applying the Threshold Propagation Strategy on the “clustered data, clustered distribution, $dist = 10$ ” setup in that we varied the threshold τ ($\tau = 0$ means that updates are spread immediately). The average global error is defined as the total change rate τ_T (Equation 6) averaged over the whole simulation and over all peers. A change size of $x\%$ means that the change in each dimension of a data item is limited to at most $x\%$ of the total dimensional range ($[0, 1000]$). Low frequency means that in each time step only 10 data items in the network

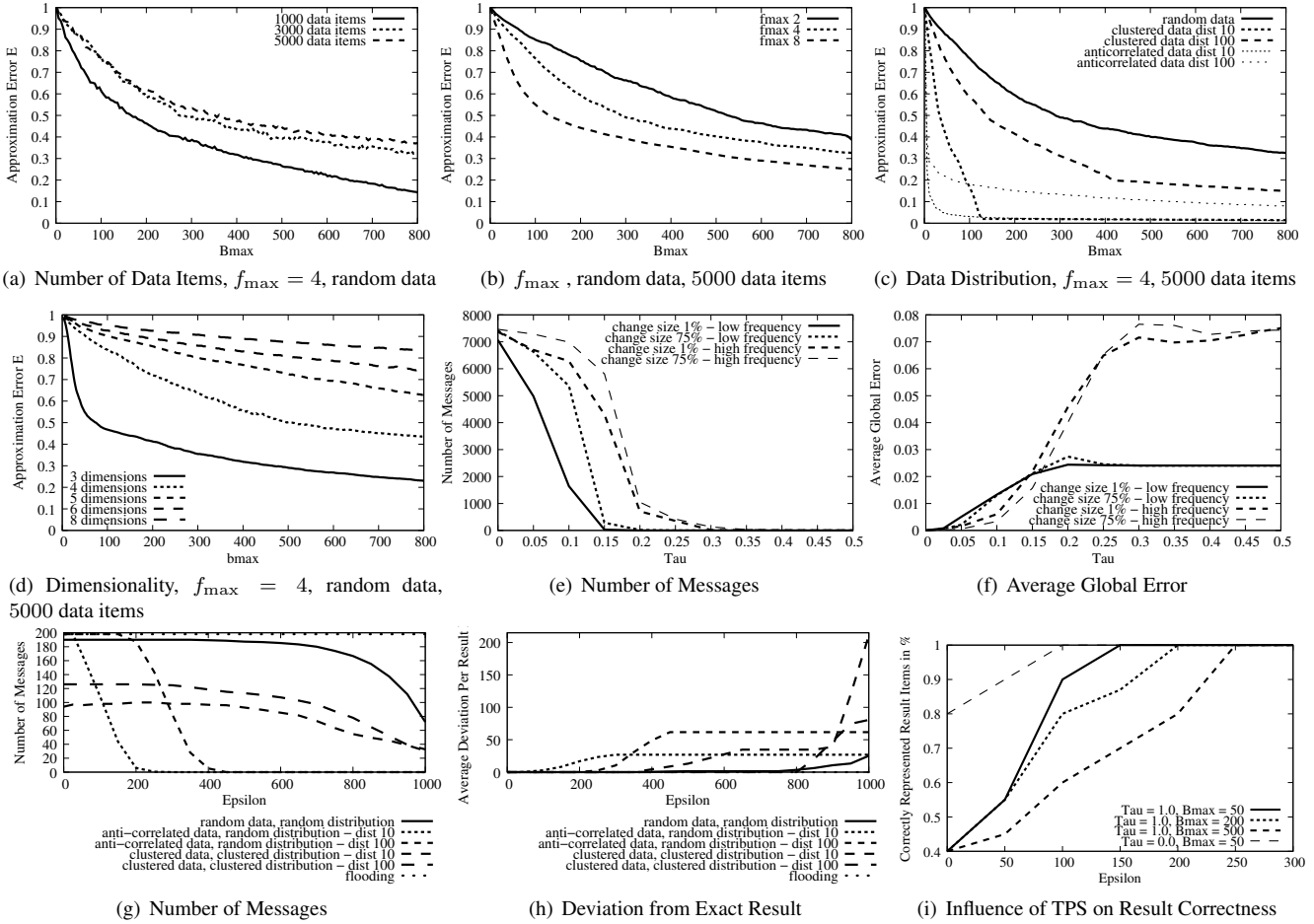


Figure 5. Influences on Approximation Quality (a-d) / Performance of TPS(e,f) / Distributed Processing of Relaxed Skylines(g-i)

are updated and the total number of updates is limited to 250 (5% of all data items in the network). High frequency means that there are 40 updates per time step and the total number of updates is 1000 (20%). Figure 5(e) shows the number of messages that is necessary to spread the updates through the network. The general tendency is quite clear: (i) the higher the change rate and the change frequency the higher is the corresponding number of messages and (ii) the higher the threshold τ the less messages are sent. Figure 5(f) reveals the consequence for higher values of τ : the error increases because less updates are spread.

6.3 Distributed Processing of Relaxed Skylines

We stated that a relaxed skyline could help to reduce the execution costs, so the higher the relaxation defined by ϵ , the higher should be the savings. Figure 5(g) shows our results for varying ϵ in the setups introduced in the beginning of this section. First, in comparison to a simple flooding of the network using QSummaries as routing indexes reduces

execution costs. Second, in the case of “Random Data, Random Distribution” the reduction is not as big as for the other setups because in such a scenario it is hard to find local representatives for a small ϵ that allows for neglecting the data of many other peers. Consequently, many peers have to be asked. In comparison of $dist = 10$ to $dist = 100$ it stands out that the higher $dist$ the more peers have to be asked. The explanation is the same as above: for a higher deviation between the data items a higher ϵ is necessary in order to represent them. The basic tendencies for all setups are pretty much the same: the higher ϵ the less peers have to be asked and thus the number of messages is reduced.

Figure 5(h) shows the average deviation (Euclidean distance) of each exact result item ($\epsilon = 0$) from the corresponding relaxed result item in dependency on the chosen ϵ . Of course, flooding the network always leads to a deviation of 0 since it always leads to the exact result. The deviations for the other setups are also relatively small in comparison to the allowed one (ϵ). In the “clustered data, clustered distribution” setups the average deviation increases only for

high ε . This is because each peer maintains a data cluster as local data. In order to represent one neighbor with a local data item the deviation must at least be the minimum distance between these clusters. When we compare the difference between the setups for $dist = 10$ and $dist = 100$ then in the cases for higher values for $dist$ the deviation is lower as for small values. The reason for this effect is *not* that bigger clusters are easier to represent. The reason is that more peers are asked (see Figure 5(g)). Thus, less data is represented which results in a lower deviation.

Note our algorithm does not commit any “error”. It only accepts a deviation between the returned result and the exact one. The algorithm quantifies this deviation when returning the result (in terms of representatives and regions).

6.4 Influence of TPS on Query Processing

Finally, let us analyze the influence of TPS on query processing. The effect that we expect is that the result quality should decrease when updates are not spread at once, i.e., if $\tau \neq 0$. For our tests we used the “clustered data, clustered distribution” setup with $dist = 10$. The updates are predefined: at the same time when the query is issued a quarter of the hundred peers crashes. These peers are those that provided the skyline result items or data nearby. Since we do not want to examine the influence of the topology and to avoid problems of network coherence, we let those peers simply propagate the “loss” of all their local data.

Figure 5(i) shows the correctness of the retrieved result R – measured as the percentage of correctly represented result items of the correct result R_C (determined by flooding). This means a result item $c \in R_C$ is correctly represented if there is any $r \in R$ for that $d(r, c) < \varepsilon$ holds. Basically, Figure 5(i) supports our anticipation that the correctness suffers from higher τ . Furthermore, the more we relax the query the higher is the chance of having represented all data items correctly even when indexes are not up-to-date. To our astonishment we also found that the correctness is also influenced by b_{max} : the higher the number of buckets, i.e., the more precise the index information the worse the correctness of the result. At first glance, this seems to be paradox. But the explanation is quite simple: the higher the number of buckets, the smaller the buckets’ sizes. And the smaller the MBBs the more precisely is our routing strategy, i.e., the less peers are asked. But if we ask less peers the chance of retrieving some “backup” result items sinks. These could contain the correct result items of the currently available data.

7 Conclusion

Routing indexes for summarizing data stored at distant nodes are an important technique for efficient query routing in PDMS. However, the benefit of enabling a semantic rout-

ing of queries comes along with a higher effort for maintenance, i.e., keeping them consistent. In this paper, we have investigated the construction and maintenance of such distributed data summaries as a problem of data replication. As a concrete example we have introduced a novel kind of DDS called QSummary which is based on hierarchical and multidimensional histograms and supports approximate query operations such as skyline queries. Furthermore, we have presented an epidemic, threshold-based update propagation strategy ensuring a certain level of freshness but restricting the update propagation overhead. In future work, we plan to combine this propagation strategy with piggybacking and query feedback approaches [7] in order to further reduce the number of update messages.

References

- [1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Puceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *ACM SIGMOD Record*, 32(3), 2003.
- [2] D. Barbará, W. DuMouchel, C. Faloutsos, P. Haas, J. Hellerstein, Y. Ioannidis, H. Jagadish, T. Johnson, R. Ng, V. Poosala, K. Ross, and K. Sevcik. The New Jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.
- [3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–432, 2001.
- [4] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS 2002*, pages 23–32, July 2002.
- [5] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massouli. From epidemics to distributed computing. *IEEE Computer*, 37(5):60–67, May 2004.
- [6] J. Gray, P. Helland, P. O’Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD 96*, pages 173–182, 1996.
- [7] M. Karnstedt, K. Hose, E.-A. Stehr, and K.-U. Sattler. Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems. In *IDEAS 2005*, pages 223–228, 2005.
- [8] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT 2004*, pages 29–47. Springer-Verlag Berlin Heidelberg 2004, 2004.
- [9] Y. Petrakis, G. Koloniari, and E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P*, pages 16–30, 2004.
- [10] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.
- [11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM 2001*, 2001.
- [12] Y. Saito and M. Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, 2005.
- [13] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM 2001*, pages 149–160, 2001.