KTV-TREE: Interactive Top-K Aggregation on Dynamic Large Dataset in the Cloud

Yuzhe Tang
Syracuse University
ytang100@syr.edu

Ling Liu
Georgia Tech
lingliu@cc.gatech.edu

Junichi Tatemura, Hakan Hacigumus
NEC Labs America
{*tatemura, hakan*}*@nec-labs.com*

Abstract

This paper studies the problem of supporting interactive top-k aggregation query over dynamic data in the cloud. We propose KTV-TREE, a top-K Threshold-based materialized View TREE, which achieves the fast processing of top-k aggregation queries by efficiently materialized views. A segment tree-based structure is adopted to organize the views in a hierarchical manner. A suite of protocols are proposed for incrementally maintaining the views. Experiments are performed for evaluating the effectiveness of our solutions, in terms of query accuracy, costs and maintenance overhead.

1 Introduction

Scenario Big data in the Cloud has recently pushed the proliferation of various scalable data management systems, including MapReduce-style batch processing systems [7, 10] and various key-value stores [8, 6]. While Map Reduce based systems (e.g. Hive [14] and Pig Latin [11]) can support analytical queries with batch processing (i.e., OLAP) and scalable key-value stores support simple interactive queries, an important type of workload that lacks support in Cloud is the online analytics. On one hand, the batch processing systems can support complex analytical queries, but with very long latency that can not be tolerable to the (impatient) online users. On the other hand, the scalable key-value stores, while providing stringent latency guarantees (e.g., in Dynamo[8]), can not process complex analytical queries, but rather simple Put/Get's.

Problem Interactive real-time analytics become more and more important for processing large dynamic dataset in the Cloud (a.k.a the Big Data). Such query allows an average end user to perform complex analytical queries against the evolving data in the Cloud and to extract timely insights of the fast changing world (e.g., for immediate decision making). Towards the goal, this work particularly addresses one important type of the analytical queries, namely the top-k aggregations with range predicate (or top-k aggregation for short). Given a table including an aggregation attribute (denoted by A), the top-k aggregation query groups all the records based on attribute A and then ranks the group-wise result based on group size before including the top-k records in the final result. In addition, the top-k aggregation allows for a range predicate on an other attribute, denoted by P. The top-k query that we target on this work can be formalized in the SQL query template in Listing 1.

Top-k aggregation queries are widely applicable to various Cloud application scenarios, including social event stream mining, enterprise network monitoring and many others. We use an example of a social web application in Yelp or Foursquare alike website. Users who log into the Yelp-like social network are interested in queries like "what are the top-5 popular restaurants in my neighborhood?" To answer such query, our top-k template 1 can be used. Consider the social web server maintains a table of user check-in events, which maintains the name of restaurant where a check-in event occurs and the location of the restaurant. By choosing Attribute A to be the restaurant name and Attribute P to be restaurant location, one can process such queries.

Listing 1: Query template of top-k aggregation with range predicate

```
SELECT A, COUNT(*) AS C FROM basetable
WHERE P IN arg.range
GROUP BY A
ORDER BY C LIMIT arg.k
```

Baselines One of the conventional strategies to evaluate top-k aggregation is the ad-hoc query processing. To be specific, the query engine proceeds with selecting based on range predicate on Attribute P, which produces the intermediate results on which various existing top-k aggregation algorithms [9, 5] can be applied. This approach, ad-hoc in nature, works well until the data in the range selection becomes large enough that top-k aggregation is an expensive operation and can not provide interactive performance. In the worst case, when user want

to have some insight about the global dataset (which means no range predicate is specified), this processing approach may end up with query broadcasting to every node in the Cloud, rendering it unable to scale out in the Cloud. Another traditional approach is to build materialized views to answer such top-k queries. However, due to the range predicates of arbitrary granularity, this approach may involve a large number of views for different range sizes. This incurs extra high overhead for view maintenance, which is impractical particularly for a streaming data source where data is generated continuously. As a result, existing work for interactive query processing in the Cloud primarily focuses on non-aggregatible OLTP workloads (e.g. range query and secondary attribute lookup [12]) and aggregation-oriented queries (e.g. top-k aggregation) are classified to be unsuitable for interactive support [3].

Proposed approach In this paper, we address the challenging problem of processing top-k aggregation queries on frequently updated dataset with interactive performance. We propose KTV-TREE, an adaptive data structure for interactive processing of top-k query over dynamic data in the Cloud. In the design of KTV-TREE, we apply the idea of selective materialized view in several aspects. On the data write path, we materialize the views for only those large enough ranges (of Attribute P) on which ad-hoc query processing can not deliver interactive latency. For each materialized view node in KTV-TREE, we only materialize the part of data that has potential to show up in a top-k result, saving the maintenance costs of data irrelevant to the final result. The KTV-TREE is organized in a binary tree, in a similar way to the distributed segment tree [15], in which parent node maintains top-k results merged from its children and the root covers the whole domain of Attribute P. By using the hierarchical tree structure to materialize multiple views, query processing can be made efficient. Typically, for a range that spans s leaf nodes, the actual number of nodes that needs to be involved in query processing is $2\log(s)$, due to the segment tree based structure[4]. The significantly reduced number of querying nodes ensures interactive query performance.

2 KTV-TREE Structure

KTV-TREE is a distributed materialized view structure built on top of a partitioned data table. As shown in Figure 1, base data table is partitioned and distributed to four machines in the Cloud. On each machine, a local KTV-TREE is constructed. The leaf node of the local KTV-TREE is the largest range on which a scan operation can finish without violating the latency SLA requirement. By having such coarse-grained leaf, rather than per-record leaf, one can save the maintenance cost of too many nodes during the data write time, without sacrificing the query latency. The upper level tree nodes maintain view of data cross multiple machines, and we call them the global view nodes. Each global view node is materialized as a hidden table in the storage server. Figure 1 shows the schema of such internal tables: given the base table of attributes A and P, the view table maintains two columns, Attribute A and Attribute C. The extra attribute C maintains the number of occurrence of the Attribute-A value in the corresponding partitions of the base table. Unlike the traditional materialized views, our view nodes in KTV-TREE are partial in two senses: 1) The view node does not maintain all the attribute-A values covered by this tree node, only attribute-A values that appear more than predefined times, denoted by O, are bound to show up in the view. 2) Each view entry is associated with a range, rather than an exact value, in Attribute C. For instance, in Figure 1, the KTV-TREE can guarantee that the parent view does have all the records whose occurrences are no less than 15, and the entry for $g2$ has its exact value of attribute C falling in range $[12,19)$. This range is obtained from merging the two children's views, which will be described in detail in the next section.

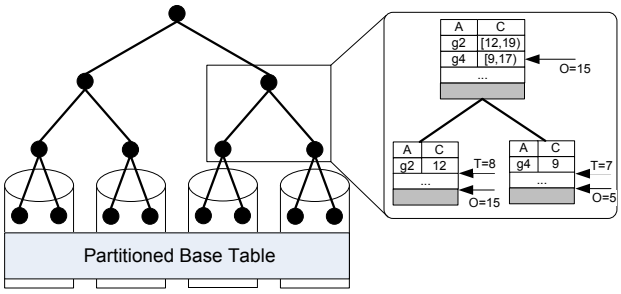


Figure 1: KTV-TREE structure

2.1 Maintaining KTV-TREE

In the presence of continuous data updates, the KTV-TREE is maintained by two processes: an online process that updates the materialized view upon each data update based on some data-dependent metadata or thresholds, and an offline process that runs in background or in off-peak hours to adjust the metadata to adapt to the changing workload.

2.1.1 Online View Maintenance

Given a data update to the base table, the online view maintenance protocol updates the KTV-TREE. To maintain a partial view, each view node keeps a threshold value T. The value of T determines whether or not an update on this view node should be further reported to its parent. Value T and O are closely related; A cutoff value O depends on the thresholds from children, that is,

$$e.O = \sum_{e' \in e.\text{children}} e'.T \tag{1}$$

The intuition is that if an entry's absent in view e, its count value C must be smaller than $\sum_{e' \in e.\text{children}} e'.T$. Otherwise,

there will be at least one entry on a child node (say e') whose count value is bigger than $e'.T$. For example, in Figure 1, entry $g2$ in root node only appears in its left child but not right child. Thus, the most the root can tell from its right child about entry $g2$ is its count value should be below the threshold $T_r = 7$. By combining exact count value 12 from left child and uncertain range $[0,7]$ from right child, the root can obtain the merged range, $[12+0, 12+7] = [12,19]$.

The online view maintenance process works in a bottom-up fashion. For a data update to the base table, it first applies to the leaf node and then iteratively propagated upwards until it's suppressed by threshold value T or it reaches the root node. Algorithm 1 illustrates the process. For data update d, view node e reads the existing local entry of the same value in attribute A (i.e., entry g with $g.\mathsf{A} = d.\mathsf{A}$), applies the update on $d.\mathsf{C}$ to the entry and persists the updated entry, say g', back to the storage. Afterward, it checks the updated count value (i.e., $g'.\mathsf{C}$) for potential propagation to node e's parent node. In particular, if either the original count value $g.\mathsf{C}$ or the updated count value $g'.\mathsf{C}$ is bigger than the local threshold $e.T$, the update d will be updated and further propagated to the parent of node e. Otherwise, the update is held back from further propagation. Notice that here we use a configurable parameter, *representative ratio* $r1$ to compare a range with a point value (i.e., T). Given a range $[l,u)$, $r1$ determines a representing point value in the range $l + (u-l)\cdot r1$, and uses this value to compare with other point value.

Algorithm 1 updateEntry(update d, viewNode e, ratio $r1$)

```
1: Entry g ← find local entry of attribute A = d.A
2: Entry g' ← g.updateCount(d.C)
3: Persist g' in d's local store
4: if compare(g, e.T, r1) > 0 || compare(g', e.T, r1) > 0
   then
5:    Change update d to u' based on e.T.
6:    updateEntry(u', e.parent, r1)
```

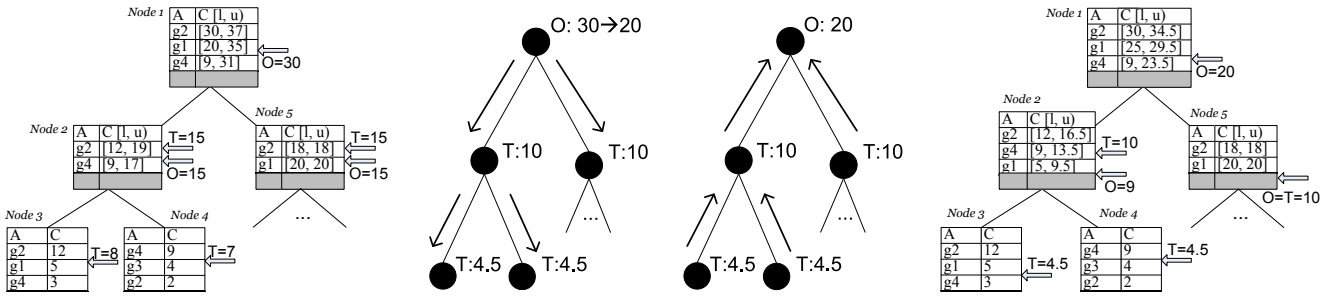
2.1.2 Offline Threshold Maintenance

To dynamically adapt the thresholds to the changing dataset, we propose an offline protocol which updates thresholds periodically. The threshold maintenance protocol works in two stages, first to update thresholds in a top-down fashion and then to bring views up to date by a bottom-up process. The first stage is illustrated in Algorithm 2. Each view node e, notified by its parent node, will first update its local threshold $e.T$ to T_1 (which is decided by the parent node). The cutoff value $e.O$ is then updated accordingly. Based on $e.O$, children's thresholds are assigned by function *splitThreshold* as in Line 6. Here different strategies can be adopted as long as Equation 1 holds; the most naive one is to split $e.O$ evenly and to uniformly distribute it to each child. Recursively, the new threshold is applied on child nodes, and the process proceeds until the leaf nodes. After this top-down stage, another process proceeds from bottom up to bring all the view nodes up to date and to match the recently updated thresholds. For local entries that are smaller than old threshold T_0 and bigger than new threshold T_1 (in case $T_1 < T_0$), we need explicitly report them to the parent to refill the view, because these entries, previously suppressed by old threshold, are no suppressed any more and should be reported. This process applies repeatedly until the root node is reached.

Algorithm 2 updateThreshold(viewNode e, Threshold T_1, ratio $r2$)

```
1:  T0 ← e.T
2:  e.O ← T1
3:  T2 ← the k-th biggest count value in the local store
4:  e.O ← min(T1, T2)
5:  for all e' ∈ e.children do
6:     T_e' ← splitThreshold(e.O, e')
7:     updateThreshold(e', T_e', r2)
8:  if T0 > T1 then
9:     for all Local entry g whose count value is between T0
       and T1 do
10:       Generate update d based on g and threshold T1
11:       Apply update d to e.parent locally (w.o. contacting
          the grandparent).
```

An example Figure 2 illustrates an example of running the threshold maintenance protocol. It considers a two-level KTV-TREE for top-2 results. The representative ratio $r1$ is 0 and thus the lower bound of a range is used. Initially, there is a mismatch between the threshold and actual entry distribution in Figure 2a. For instance, at node 2, the second entry (i.e., $g4$) has a count value with lower bound 9, while the cutoff value is 15, implying certain desired tuples may be missed. This is the motivating scenario to run the threshold-update process in KTV-TREE. In the first top-down stage, the root node, after checking the local entries, updates its cutoff value to be 20, which is the lower bound of its top-2 local count values. Then, the root node notifies its children to reset threshold by splitting the local cutoff value evenly (i.e., value 10) among its children nodes. Child node 2 then checks its local store and finds the lower bound of top-2 count values is 9 which is smaller than threshold 10. Then node 2 updates its cutoff value to 9 and similarly triggers its children to reset thresholds recursively. At the end of the first stage, all thresholds are set but without updating entries there will be inconsistency. For example, for node 3 whose threshold changes from 8 to 4.5, its second entry of count value 5 which was previously suppressed by old threshold now should be propagated to parent node. In general, our second stage protocol will "refill" the entries due to threshold drops at some view nodes. Figure 2c illustrates the bottom-up process. The final state, shown as in Figure 2d, holds the property that each tree node's entries would fully cover the true top-k result after the threshold maintenance protocol.



(a) Initial state (b) Stage 1: Top-down process to reset threshold (c) Stage 2: Bottom-up process to refill entries (d) Final state

Figure 2: An example of updating threshold in KTV-TREE

2.2 Query Rewrite and Evaluation

Given a top-k query whose range covers s leaf nodes, the KTV-TREE can route the query to $2\log(s)$ nodes, due to a property provided by segment tree. On each of the view nodes, our query engine rewrites the query which was originally written against the base table schema to a form that matches the view schema. To be specific, the query template in Listing 1 is rewritten to the form in Listing 2. After results are retrieved from each view node, they are merged and joined together to form the final result. Specifically, entries of the same attribute A but from different view nodes have their attribute-C values summed up. Then all the entries are sorted by the global C, and a final top-k list is produced.

Listing 2: Rewritten query against a single view table

```
SELECT A, C FROM viewtable
ORDER BY C LIMIT arg.k
```

3 Implementing KTV-TREE on Key-value Stores

We demonstrate a KTV-TREE prototype built on top of a key-value store for Cloud web applications. We have implemented the prototype on HBase [1][1], a widely used scalable key-value store modeled after Google's BigTable [6]. There are two designs we address in our implementation: 1) the mapping of logic KTV-TREE to physical region servers in HBase, and 2) the implementation of KTV-TREE maintenance and query evaluation protocols.

- To map KTV-TREE to HBase, the most straightforward way is to implement each view node as an HBase table and leave the mapping to HBase's mechanism of table sharding and distribution. This approach may undermine the data locality. In our prototype, we implement the local KTV-TREE as local in-memory structure co-located with the base table partition and implement the global KTV-TREE as regular HBase tables. The mapping of global KTV-TREE is manually set and follows our prior work [13] to achieve a one-to-one mapping from internal nodes to base able partitions. Specifically, any internal node in the global KTV-TREE, be it a left/right child of its father, is mapped to the rightmost/leftmost leaf node of the sub-tree rooted at this node. The root node is mapped to the leftmost leaf node. This mapping implies load balancing; extra maintenance workload of KTV-TREE is evenly distributed to all involving table partitions.

- To implement the online view maintenance protocols, we inject code into the write path of HBase by using Co-Processor [2] which is a set of programming hooks and allows for associating events with HBase actions (e.g., reads and writes). Specifically, each update to a base table entry synchronously invokes the online view maintenance routine in Algorithm 1. The offline view maintenance protocols are implemented by disabling all online operations and running a batch process to rebuild thresholds.

4 Experiments

In this section, we demonstrate an implementation of KTV-TREE and show the experimental results. The experiments are concerned on the maintenance overhead, query accuracy and the scalability.

4.1 Experimental Setup

Dataset used in the experiments is $20,000$ updates divided into two batches: The first batch of $10,000$ updates is used to load the system and to initialize the thresholds. The second batch of $10,000$ updates is used for experimental evaluation. Each update is a triplet, following the schema of the view table $< A, B, C >$. The value of attribute A, is randomly picked from 50 distinct tags. Attribute B is a numeric value, randomly distributed in domain $[0, 32]$. The updates on attribute C are bounded by 50. In the experiments, different data distributions are used to generate update values on attribute C,

[1] We use version 0.94.2.

including uniform distribution and Zipf distributions (by default). In the experiment, we used twenty nodes [2] and mainly test the in-memory KTV-TREE (i.e., without testing disk accesses).

The primary experiment parameter that we vary is the representative ratio. Recall that given a range R with lower bound l and upper bound u, the value specified by the ratio r is then $l + (u - l) \cdot r$. This representative ratio is used for two purposes; one ratio, denoted by $r1$, is to sort the list and provide a top-k view (by comparing two ranges), and the other, denoted by $r2$, is to keep a small update from being propagated to the parents (by comparing a range and a threshold).

4.2 Update Costs

Updating KTV-TREE is an iterative process which affects one leaf node and its multiple ancestors. The costs are then measured by the number of tree nodes affected by the update. First, we vary the representative ratio r and record the update costs, as in Figure 3. The costs of our KTV-TREE increases as representative ratio goes up. Comparing to the baseline approach, the maximal cost saving can reach around 40%.

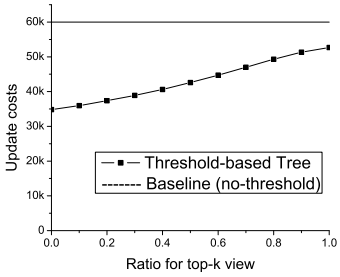


Figure 3: Update costs

4.3 Accuracy

For a top-k query, the accuracy is measured by the fraction of true top-k results. For example, half of entries in current result being true top-k entries yields a query accuracy of 50%. By our query model, different queries could span different number of tree nodes, and in this experiment, we only test the query that can be answered by a single tree node. Specifically, two cases are tested; one is accuracy of query that span whole domain (of attribute B), or the accuracy of the root node. And the other is accuracy averaged across all tree nodes. Results are plotted in Figure 4a. Generally, accuracy in both of these cases show similar trend along with representative ratio changes; they reach the peak when representative ratio is 0.6. The root node has smaller accuracy comparing to tree nodes at other levels. For example, root node has its highest accuracy to be 0.9, which is lower than average accuracy of all tree node with all possible values on r. This is mainly due to that view entries at root node need go through multiple rounds of filtering processing, incurring higher imprecision.

We further explored for a detailed view, that is, how tree nodes at different levels are affected by different representative ratios. We differentiate $r1$ and $r2$ in this experiment and vary them independently. Results are shown as in Figure 4b and Figure 4c, for accuracy of root and that of all nodes, respectively. In each figure, x axis and y axis represent the ratio for top-k view and the ratio for filtering top-k views. As it shows, the accuracy of high level internal nodes (like root) is more sensitive to the ratio for top-k views, while the accuracy of low level internal nodes (which dominate all tree nodes) is more sensitive to the ratio for filtering. Rule of thumb is that both of these ratios should be set to 0.8.

4.4 Scalability

In this section, we test the scalability of the system when domain of attribute A grows large. We fixed the total number of updates to be $10,000$, but changes the domain of attribute A. The results in terms of accuracy and update costs are reported as in Figure 5. When the domain of attribute A becomes large, the overall update costs drops while the accuracy largely stays the constant.

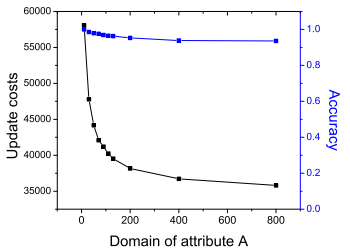


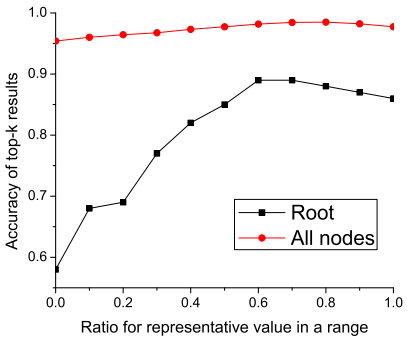
Figure 5: Scalability to domain size of attribute A

5 Conclusion

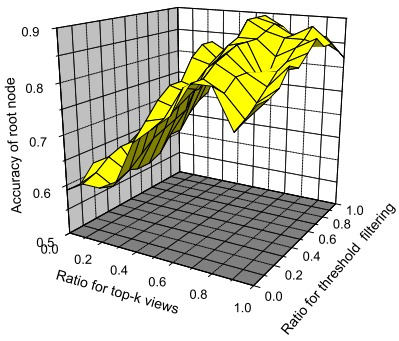
In this work, we study the problem of supporting interactive top-k aggregation query over dynamic data in the cloud. We propose KTV-TREE, a top-K Threshold-based materialized View TREE, which achieves the fast top-k aggregation processing by efficiently materialized views. A segment tree-based structure is adopted to organize the views in a hierarchical manner. A suite of protocols are proposed for incrementally maintaining the views. We demonstrate the feasibility of KTV-TREE by building a prototype on key-value stores. Experiments are performed for evaluating the effectiveness of our solutions, in terms of query accuracy, query costs and maintenance overhead.

Acknowledgments The first and last authors are partially supported by grants from NSF CISE NetSE, NSF CyberTrust, an IBM SUR grant, an IBM faculty award, and a grant from Intel Research Council.

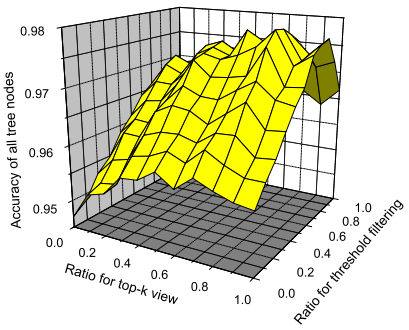
[2]The machine specification includes a 2.4 GHz 64-bit Quad Core Xeon processor (with hyper-threading support) and 12 GB RAM.



(a) With different representative ratios



(b) Accuracy of root node



(c) Accuracy of all tree nodes

Figure 4: Accuracy of top-k query results

References

[1] http://hbase.apache.org/.

[2] https://blogs.apache.org/hbase/entry/coprocessor_introduction.

[3] ARMBRUST, M., CURTIS, K., KRASKA, T., FOX, A., FRANKLIN, M. J., AND PATTERSON, D. A. Piql: Success-tolerant query processing in the cloud. *PVLDB 5*, 3 (2011), 181–192.

[4] BERG, M. D., CHEONG, O., KREVELD, M. V., AND OVERMARS, M. *Computational Geometry: Algorithms and Applications*, 3rd ed. ed. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

[5] CAO, P., AND WANG, Z. Efficient top-k query calculation in distributed networks. In *PODC* (2004), pp. 206–215.

[6] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI* (2006), B. N. Bershad and J. C. Mogul, Eds., USENIX Association, pp. 205–218.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 205–220.

[9] FAGIN, R., LOTEM, A., AND NAOR, M. Optimal aggregation algorithms for middleware. In *PODS* (2001).

[10] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), P. Ferreira, T. R. Gross, and L. Veiga, Eds., ACM, pp. 59–72.

[11] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference* (2008), pp. 1099–1110.

[12] TANG, Y., IYENGAR, A., TAN, W., FONG, L., LIU, L., AND PALANISAMY, B. Deferred lightweight indexing for log-structured key-value stores. In *IEEE/ACM 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen Guandong, China, May 2015* (2015).

[13] TANG, Y., ZHOU, S., AND XU, J. Light: A query-efficient yet low-maintenance indexing scheme over dhts. *IEEE Trans. Knowl. Data Eng. 22*, 1 (2010), 59–75.

[14] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTHONY, S., LIU, H., AND MURTHY, R. Hive - a petabyte scale data warehouse using hadoop. In *ICDE* (2010), pp. 996–1005.

[15] ZHENG, C., SHEN, G., LI, S., AND SHENKER, S. Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)* (2006).