# Rank Join Queries in NoSQL Databases

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK

nikos.ntarmos@
glasgow.ac.uk

Ioannis Patlakas
Max-Planck-Institut für
Informatik, Germany

patlakas@mpi-inf.mpg.de

Peter Triantafillou
School of Computing Science
University of Glasgow, UK

peter.triantafillou@
glasgow.ac.uk

## ABSTRACT

Rank (i.e., top-$k$) join queries play a key role in modern analytics tasks. However, despite their importance and unlike centralized settings, they have been completely overlooked in cloud NoSQL settings. We attempt to fill this gap: We contribute a suite of solutions and study their performance comprehensively. Baseline solutions are offered using SQL-like languages (like Hive and Pig), based on MapReduce jobs. We first provide solutions that are based on specialized indices, which may themselves be accessed using either MapReduce or coordinator-based strategies. The first index-based solution is based on inverted indices, which are accessed with MapReduce jobs. The second index-based solution adapts a popular centralized rank-join algorithm. We further contribute a novel statistical structure comprising histograms and Bloom filters, which forms the basis for the third index-based solution. We provide (i) MapReduce algorithms showing how to build these indices and statistical structures, (ii) algorithms to allow for online updates to these indices, and (iii) query processing algorithms utilizing them. We implemented all algorithms in Hadoop (HDFS) and HBase and tested them on TPC-H datasets of various scales, utilizing different queries on tables of various sizes and different score-attribute distributions. We ported our implementations to Amazon EC2 and "in-house" lab clusters of various scales. We provide performance results for three metrics: query execution time, network bandwidth consumption, and dollar-cost for query execution.

## 1. INTRODUCTION

Cloud stores have become the storage of choice for a large variety of big data producers, consumers, and managers (e.g., Twitter, Facebook, Google, Amazon, etc.) For many modern Big Data applications, RDBMSs were found lacking, particularly with respect to scalability (in terms of number of data items, users, operations per second, etc.), despite valiant efforts (e.g., sharding, memory caches, partial denormalization, etc.). To fill this gap, two comple-

mentary technologies emerged: NoSQL databases and the MapReduce framework. Interestingly, even traditional key RDBMS players, such as Oracle, are now focusing on NoSQL products coupled with MapReduce platforms. There is an impressive list of NoSQL cloud databases; e.g., BigTable, DynamoDB, Riak, HBase, Cassandra, etc. All these are purpose-built to scale across a large number of servers (by sharding/horizontal partitioning of data items), to be fault tolerant (through replication, write-ahead logging, and data repair mechanisms), to achieve high write throughput (by employing memory caches and append-only storage semantics) and low read latencies (through caching and smart storage data models), flexibility (with schema-less design and denormalization), and development friendliness (Object Relational Mappings are typically avoided, etc.). Further, different systems offer different approaches to issues such as consistency, replication strategies, data types, and models.

The data model employed by NoSQL DBs can be viewed as a key-value model, built around four core abstractions: (a) the "key-value pair": a quadruplet {key, column name, column value, (and perhaps a timestamp)}, uniquely identified by the combination of key, column name (and timestamp, where applicable); (b) the "table": an ordered collection of key-value pairs; and (c) the "row": a collection of all key-value pairs in a given table, sharing the same key. Some systems further employ the notion of "column families", in essence partitioning the table data vertically so that each such family includes key-value pairs with specific column names. All such systems support efficient point queries and sequential scans on key-value pairs and rows based on their key, as well as efficient insertion/deletion of key-value pairs. However, queries on other parts of the data (e.g., column names/values, timestamps, etc.) are costly, often requiring a scan of all key-value pairs even for simple equality queries.

Despite original intents, NoSQL is now "re-baptized" to spell "Not Only SQL". As more data poured into these systems, the need for complex queries – particularly for analytics – emerged, leading to the rise of data warehousing systems, such as Hive[24] and Pig[21], offering SQL-like interfaces. This went hand-in-hand with the realization of the shortcomings of denormalization for several real-world analytic workloads; carrying denormalization to the extreme implies a huge amount of data being repeated across very large numbers of rows in a "universal" table, creating an update/maintenance nightmare and utterly negating several of the key performance advantages of NoSQL systems in the long run. Although NoSQL guarantees fast keyed-row retrievals, these rows now contain typically lots of data use-

less to most queries, resulting in poor query performance as many more disk IOs are needed to get the desired data (e.g., in typical queries involving large scans of rows), and more useless data are shipped across the NoSQL cloudstore substrate. Hence, inevitably NoSQL systems transformed to contain several tables, linked through foreign-key-like constructs (although without the consistency semantics inherent in classic RDBMSs) and joined at query time. The issue is that the burden to do this lies on the applications. This is currently the conventional wisdom, especially when modeling many-to-many relationships and storing them in NoSQL systems. Orthogonally, emerging analytic tasks often rely on data dispersed across different files or tables.

*Motivations.* As joins are very resource hungry, several recent research efforts have attempted to expedite them in cloud stores. So far, ranked (top-$k$) equi-joins have been completely overlooked in this setting, despite the fact that these queries arise naturally in a variety of situations, as joining and ranking are fundamental to many analytics tasks. Take for example a collection of per-day search engine logs, consisting of phrases and their frequency of appearance in user inputs, with a separate table or file per day. Now imagine we wish to find the $k$ most popular phrases appearing in several of these days. This would be formulated as a rank-join query, where the phrase text is the join attribute, and the total popularity of each phrase is computed as an aggregate over the per-day frequencies. Rank equi-joins also arise in full-text search scenarios. Imagine a collection of posting lists over a large text corpus consisting of several documents. Each posting list would include information for documents containing the related keyword, with each list entry consisting of (at least) the document identifier and the document's relevance score with regard to the keyword. With the size of many of these lists being in the gigabytes even for relatively small collections such as Wikipedia dumps (and much larger when scaling to web archives and/or the actual World-Wide-Web), and given the need to scan through them efficiently, it is only reasonable to assume that each list is stored in a separate table in a key-value store. Then, finding the most relevant documents for two (or more) keywords, consists of a rank-join over the corresponding posting lists, where the document ID is the join attribute and the relevance of each document to the search phrase is computed using a function over the individual relevance scores.

## 1.1 Problem Formulation

We assume operation in a distributed shared-nothing cloud store. Please note that the raw table data can be stored either at the filesystem level (e.g., HDFS) or at any NoSQL store. Our algorithms are impervious to this!

Formally, a top-$k$ join query can be written as:
    SELECT *select-list* FROM $R_1, R_2, \ldots, R_n$
    WHERE *equi-join-expression*$(R_1, R_2, \ldots, R_n)$
    ORDER BY $f(R_1, R_2, \ldots, R_n)$ STOP AFTER $k$
Scoring of individual rows is typically based on either a (predefined) function on attribute values, or some explicit "score" attribute (e.g., movie rating, PageRank score, etc.). The solutions we discuss make no distinction between these two cases and work equally well for both; however, for ease of presentation and without loss of generality, we assume there is a scoring attribute in each row. Furthermore, for ease of presentation, we assume that score attributes take

values in $[0, 1]$; it should be obvious that our algorithms perform the same with arbitrary score values as well, provided that there is a total ordering on these values. Last, as is common in rank-join works, the score of tuples in the join result set is computed using a monotonic aggregate function $f(\cdot)$ on their individual scores.

A naive approach would first compute the join result, then rank and select the top-$k$ tuples; this is the approach of Hive and (with minor optimizations) of Pig. This is obviously extremely costly, even in centralized settings, let alone when data have to be shipped across the network. Our challenge is to compute the result set without producing the full join result, ensuring that the amount of data that is transferred is as little as possible and the query latency is small.

## 1.2 Contributions

A study of how to efficiently process top-k join queries in NoSQL cloudstores is very much lacking, given the rapid popularity increase of such environments and their unique characteristics. We will use as a reference point the baseline techniques of using Hive or Pig to formulate and execute such queries in a massively parallel manner. However, acknowledging their disadvantages we will contribute and study the performance of a number of different approaches. First, we contribute a MapReduce solution that is based on specialized indices. Second, in the no-MapReduce realm, we contribute an algorithm coined Inverse Score List Rank Join (ISL), which is based on the popular HRJN[13] centralized rank join algorithm. Our third contribution is an algorithm based on a novel rank-join statistical access structure, coined the Bloom Filter Histogram Matrix (BFHM). We prove that the BFHM algorithm can achieve 100% recall (despite its probabilistic nature). We have chosen to store all our access structures in a NoSQL DB. Although not a hard requirement, this choice was dictated by the need to handle possible large volumes of new item insertions/deletions, for which NoSQL DBs are much better suited than DFSs. Further, we also provide algorithms for efficiently maintaining indexes in the face of updates. Our solutions offer trade-offs with regard to various metrics. We thus further contribute an in-depth performance evaluation in real-world systems (Amazon's EC2 and in-house clusters, using TPC-H datasets), against the baseline approaches. The metrics are: query processing time, network bandwidth consumption, and query processing dollar cost (e.g., when executed on charge-per- item infrastructures, such as DynamoDB).

## 2. RELATED WORK

*Rank Queries.* Well-known ranking operators include the top-$k$ and $kNN$ operators. Top-k (selection) operators typically accept as input a set of records each of the type $< ID, score_1, \ldots, score_n >$, a monotone score aggregation function, $f(score_1, \ldots, score_n) \to [0, 1]$ and a threshold number $k$, and produce the IDs of records whose aggregate score (from all $n$ score attributes) is in the top-$k$ among the score of all records. $kNN$ operators are different; they accept as input a specific record, $r$, a table of records $T = \{t_1, \ldots, t_n\}$ and a notion of similarity among records $sim(s, t)$, and produce the result set $R = \{t_{R_1}, t_{R_2}, ..., t_{R_k}\}$ with $t_{R_i} \in T$ and $|R| = k$, so that $sim(r, t_{R_i}) > sim(r, t_j), \forall t_j \in T \bigwedge t_j \notin R$.

Our paper focuses on ranking a la top-$k$ queries defined above. A survey of top-$k$ query processing algorithms for

this setting is presented in [14]. In [10] instance-optimal algorithms are presented, i.e., the Threshold Algorithm (TA) and variants, using sorted and/or random accesses. The first notable distributed variant of TA is the Three-Phase Uniform Threshold (TPUT) algorithm[5]. KLEE[16] improved TPUT by employing histograms and Bloom filters for each node to limit the tuple search space and bandwidth consumption. Last, the Threshold Join Algorithm (TJA) [28] is a top-$k$ selection query processing algorithm, using an outer join step to maintain partial top-$k$ results as these are aggregated at parent nodes. TJA is developed for a hierarchical (sensor) topology – a setting drastically different than that of NoSQL clusters.

*Rank and Join Queries.* $kNN$ *joins*[26], as well as *similarity joins*[4] include both rank and join operators. Similarity joins accept as input two collections of sets, $R$ and $S$, a notion of set similarity and a threshold similarity value $t$, and return all pairs $(r, s)$ with $\{r \in R\}$ and $\{s \in S\}$, such that $sim(r, s) > t$. This is equivalent to returning, for each record $r$ from $R$, all records $s$ from $S$, such that $sim(r, s) > t$. $kNN$ join operators are similar to the above, generalizing the $kNN$ operator to perform $kNN$ operations not for just a given record but for a set of records. Finally, *top-k similarity joins*[27], like similarity and $kNN$ joins, use a notion of distance to define similarity among records from $R$ with those from $S$. A top-$k$ similarity join returns those joined record pairs having the highest $k$ similarities. The top-$k$ join operators considered in this paper extend top-$k$ selection queries by aggregating score attributes from two (or more) relations and by returning the top-$k$ scores only of records which are in the result of a join operation (based on some other common attribute of the two relations). As such, top-$k$ join queries are substantially different than $kNN$ join and (top-$k$) similarity join queries. Joining is performed on separate explicit join attributes (introducing challenges/opportunities for extra indexing and optimisations) and ranking is based on a monotone aggregate function of the relations' score attributes (as opposed to a distance between records).

Ilyas et al. presented NRA-RJ [12] to support such rank joins. A pipelined operator, $J^*$, was presented in [19], providing a join operator with a general non-equi-join condition. In [13], Ilyas et al. present an influential algorithm for ranked join (HRJN) (to be discussed at more length later). More recently, [22] presented the Pull/Bound Rank Join (PBRJ) algorithm, a generalization of HRJN-style algorithms. [25] proposes a join graph in which joining attributes are represented by nodes and relations among attributes by edges. This approach provides support for top-$k$ join queries (other than inner-join ones) over web databases.

*Join Queries on Clouds.* Recently join queries in the cloud received considerable attention. Hive [24] and Pig [21] support joins over very large datasets using MapReduce. HadoopDB[1] replaces local data stores with DBMS instances and supports the execution of MapReduce jobs over them; in essence, it is a parallel DBMS enriched with MapReduce capabilities, and is unlike NoSQL systems. Hadoop++[7] proposes "Trojan indices", created at data load time on join attributes and collocated with the data read by each mapper, allowing mappers to avoid expensive disk scans to sort data. Then, "Trojan Joins" co-partition the splits of the relations to be joined, yielding map-only jobs and saving

considerable overhead. Results have also been produced for other than equi-joins and 2-way joins [2, 15, 20].

*Bloom Filters, Joins, and Top-k.* A Bloom filter (BF)[3, 17] is a data structure that compactly represents a set of items as a bit vector to expedite membership queries. [6, 18] use BFs to estimate the cardinality of a join result. Our treatment of BFs differs in its statistical intuition from both [18] and [6]. Finally, KLEE also uses BFs and histograms for top-$k$ queries; however, KLEE does not deal with joins, the actual data structures and their use are different, it doesn't use counting filters, and cannot guarantee 100% recall.

*Rank Joins in Distributed Settings.* Although inspirational, none of the above works have attempted to solve the problem of top-$k$ equi-join queries in cloud stores. [29] and [8] tackled the problem of rank join processing in large distributed systems. They both attempt to compute a bound on the scores of individual tuples from the base relation, in order to prune tuples not participating in the top-$k$ join result, and both assume operation over a DHT network overlay. [29] uses a sampling stage in which the querying node multicasts the query to a random set of peers. These peers then perform a hash-join by rehashing their data onto the DHT using the join value as the hash function input. The querying node collects the result set and retains the $k$'th highest score. It then broadcasts this score to all nodes, which in turn perform a distributed hash-join again, only now limiting the rehashed items to those that can produce a join result with a score above the threshold (assuming they join with a tuple with the maximum score value). [8] employ 2-D equi-width histograms; for each of the distinct join values in the input data, they build and store a histogram on the corresponding score values. Query processing consists of two stages – score bound estimation using the histogram buckets, and pulling of data tuples with scores above the bound – repeated in sequence until the final result is produced. As maintaining one bucket per distinct join value is not feasuble in real scenarios, the authors generalize their solution by grouping same-score buckets for adjacent join values and combining them using the uniform frequency assumption. Both of these approaches (much like our ISL and BFHM), fall in the general family of PBRJ-style algorithms, interchanging between bound computation and data pulling. Both these and ISL produce bounds on the tuple scores, ignoring however their join attribute values, thus ending up transferring more tuples than necessary (as several of them may not contribute to the final result due to not joining with any other tuple). Such approaches are at a disadvantage in cloudstores, as their processing time is dominated by data transfers. This situation is further aggravated by the fact that sampling ([29]) and approximate statistics ([8]) often lead to inaccurate estimations and either low recalls (e.g., as low as 0.2 for [8]) or extremely high query processing costs (to be shown shortly). BFHM, however, combines histograms with Bloom filters to locate tuples that will indeed end up in the final result set, achieving further savings in query processing time, bandwidth consumption, and dollar-cost, while guaranteeing a perfect 100% recall.

## 3. BASELINE RANK JOINS

We focus on two-way equi-joins; extending the algorithms

| $R_1$ | | | $R_2$ | | |
|---|---|---|---|---|---|
| row key | join value | score value | row key | join value | score value |
| $r_{1_1}$ | d | 0.82 | $r_{2_1}$ | a | 0.51 |
| $r_{1_2}$ | c | 0.93 | $r_{2_2}$ | b | 0.91 |
| $r_{1_3}$ | c | 0.67 | $r_{2_3}$ | c | 0.64 |
| $r_{1_4}$ | d | 0.82 | $r_{2_4}$ | d | 0.53 |
| $r_{1_5}$ | a | 0.73 | $r_{2_5}$ | d | 0.41 |
| $r_{1_6}$ | c | 0.79 | $r_{2_6}$ | d | 0.50 |
| $r_{1_7}$ | b | 0.82 | $r_{2_7}$ | a | 0.35 |
| $r_{1_8}$ | b | 0.70 | $r_{2_8}$ | a | 0.38 |
| $r_{1_9}$ | d | 0.68 | $r_{2_9}$ | a | 0.37 |
| $r_{1_{10}}$ | a | 1.00 | $r_{2_{10}}$ | c | 0.31 |
| $r_{1_{11}}$ | b | 0.64 | $r_{2_{11}}$ | b | 0.92 |

**Figure 1: Running example: Tuples of $R_1$ and $R_2$**

to multi-way joins is straightforward. We first describe Hive's and Pig's approaches, as the baseline solutions for rank-join queries. Fig. 1 shows the input for our running example.

## 3.1 Rank Joins with Hive and Pig

In Hive, rank join processing consists of two MapReduce jobs plus a final stage. The first job computes and materializes the join result set, while the second one computes the score of the join result set tuples and stores them sorted on their score; a third, non-MapReduce stage then fetches the $k$ highest-ranked results from the final list.

Pig takes a smarter approach. Its query plan optimizer pushes projections and top-$k$ (STOP AFTER) operators as early in the physical plan as possible, and takes extra measures to better balance the load caused by the join result ordering (ORDER BY) operator. Specifically, 3 MapReduce jobs are used. The first computes the join result: mappers scan the table files, do early projections (stripping out unrelated columns), and emit rows with the join value as their key; then, reducers group together rows with the same join value, produce the join result set, and store it in an HDFS file. The second MapReduce job is a by-product of the OR-DER BY clause; it samples the records in the join result file in the map phase, and appropriate quantiles are computed at the reduce phase. These quantiles are then used to construct a balanced partitioner for the third job, which orders the temporary records on their score and produces the top-$k$ result set. First, the map phase emits the temporary records with their join score as their key, and a combiners take over producing a local top-$k$ list. These lists are then assigned to a sole reducer producing the final top-$k$ result set.

## 4. INDEXED RANK JOINS

As BigTable/HBase were the archetypical key-value cloud-stores, we borrow their terminology in the description of the various algorithms and examples. It should be clear, though, that our indices and algorithms apply (perhaps with slight, obvious modifications) to all contemporary key-value stores.

## 4.1 Inverse Join List MapReduce Rank-Join

Our first algorithm – Inverse Join List MapReduce rank join (IJLMR) – uses MapReduce, but utilizes an index to reduce the required MapReduce jobs to one, and avoid extra network transfers and inefficiencies.

### 4.1.1 IJLMR Index

In the above approaches, the first stage mappers actually create an inverted list of input tuples keyed by their join

---

**Algorithm 1** IJLMR Index Creation

```
1 Input: Rows from a single column family (e.g., A), of
        the form {row.rowKey:  row.joinValue, row.score}
2 Output: IJLMR index rows for A
3 Map():
4   foreach(row ∈ sequential scan of A)
5     emit(row.joinValue:  row.rowKey, row.score);
```

| Row key (join value) | Index tuples ({row key, score}) | |
|---|---|---|
| | $R_1$ | $R_2$ |
| a | {$r_{1_{10}}$,1.00}, {$r_{1_5}$,0.73} | {$r_{2_1}$,0.51}, {$r_{2_7}$,0.35}, {$r_{2_8}$,0.38}, {$r_{2_9}$,0.37} |
| b | {$r_{1_7}$,0.82}, {$r_{1_8}$,0.70}, {$r_{1_{11}}$,0.64} | {$r_{2_2}$,0.91}, {$r_{2_{11}}$,0.92} |
| c | {$r_{1_2}$,0.93}, {$r_{1_3}$,0.67}, {$r_{1_6}$,0.79} | {$r_{2_3}$,0.64}, {$r_{2_{10}}$,0.31} |
| d | {$r_{1_1}$,0.82}, {$r_{1_4}$,0.82}, {$r_{1_9}$,0.68} | {$r_{2_4}$,0.53}, {$r_{2_5}$,0.41}, {$r_{2_6}$,0.50} |

**Figure 2: Running example: IJLMR index table**

values; this is, in essence, a materialized view where each entry has a join value as its key, and the input rows with that specific join value as its set of values. Our IJLMR index consists of a space-optimized form of these inverted lists, where index values consist of a list of tuples each being a combination of the row key and score value of the indexed row (Fig. 2). The IJLMR index for each indexed table is stored as a separate column family in one big table. This means that, if the table is split up/sharded and distributed across the NoSQL store nodes, index entries for the same join values across all indexed tables are stored next to each other on the same node. The IJLMR index is built with a map-only MapReduce job (Algorithm 1) – a special type of MapReduce job where there are no reducers and the output of mappers is written directly into the NoSQL store. We also provide routines for online maintenance and updates to this index, to be discussed shortly.

### 4.1.2 IJLMR Query Processing

The IJLMR query processing algorithm consists of a single MapReduce job/stage, with several mappers and a single reducer (Algorithm 2). In this job, each mapper scans through its partition of the IJLMR index, reading columns from the index column families for the joined tables, one row at a time. For each row, it computes the Cartesian product (i.e., the join result) and join score of index entries from the different column families; e.g., the mapper responsible for join value $a$ (see Fig. 2) would produce $2 \times 4$ key-values, the mapper responsible for $b$ would produce $3 \times 2$ tuples, etc. The mappers store in-memory only the top-$k$ ranking result tuples, and emit their final top-$k$ list when their input data is exhausted. The single reducer then combines the individual top-$k$ lists and emits the global top-$k$ result.

In addition to reducing the overall MapReduce jobs and stages down to one, this design has the added benefit that data transfers due to MapReduce shuffling/sorting are minimized; the Hadoop framework ensures that each mapper is executed on the NoSQL store node storing its input region data (or as close to it as possible), and thus only the individual top-$k$ result sets are transferred across the network and shuffled/sorted at the single reducer. As we shall see in the performance evaluation section, this approach

**Algorithm 2** IJLMR Rank-Join

```
1 Input: Rows from the IJLMR index for column families A
       and B, of the form {row.joinValue: row.rowKey,
       row.score}
2 Output: Top-k join result set
3 Map():
4   foreach(row ∈ input) {
5     HashTable tuplesA=∅, tuplesB=∅;
6     SortedList results=∅;
7     foreach(kv ∈ row) {
8       if (kv.columnFamily == A) {
9         myTuples = tuplesA.get(kv.joinValue);
10        otherTuples = tuplesB.get(kv.joinValue);
11      } else {
12        myTuples = tuplesB.get(kv.joinValue);
13        otherTuples = tuplesA.get(kv.joinValue);
14      }
15      foreach (kv' ∈ otherTuples) {
16        results.add(innerJoin(kv, kv'));
17        results.trim(k);
18      }
19      myTuples.append(kv.joinValue, kv);
20    }
21    emit(results);
22 Reduce():
23   SortedList results=∅;
24   foreach(kv ∈ input) {
25     results.add(kv);
26     results.trim(k);
27   }
28   emit(results); // Final top-k result set
```

**Algorithm 3** ISL Index Creation

```
1 Input: Rows from a single column family (e.g., A), of
       the form {row.rowKey: row.joinValue, row.score}
2 Output: ISL index rows for A
3 Map():
4   foreach(row ∈ sequential scan of A)
5     emit(row.score: row.rowKey, row.joinvalue);
```

| Row key (score) | Index tuples ({row key, join value}) | |
|---|---|---|
| | $R_1$ | $R_2$ |
| -1.00 | $\{r_{1_{10}}, a\}$ | |
| -0.93 | $\{r_{1_2}, c\}$ | |
| -0.92 | | $\{r_{2_{11}}, b\}$ |
| -0.91 | | $\{r_{2_2}, b\}$ |
| -0.82 | $\{r_{1_1}, d\}, \{r_{1_4}, d\},$ $\{r_{1_7}, b\}$ | |
| -0.79 | $\{r_{1_6}, c\}$ | |
| ... | | |
| -0.35 | | $\{r_{2_7}, a\}$ |
| -0.31 | | $\{r_{2_{10}}, c\}$ |

**Figure 3: Running example: ISL index table**

achieves at least an order of magnitude faster query processing times compared to Hive, and several orders of magnitude less bandwidth consumption. Unfortunately, note that the mappers still have to scan through the entire input dataset, weighing on the dollar-cost of query processing, which is almost as high as that of the Hive approach.

## 4.2 Inverse Score List Rank-Join

Clearly, network and disk I/O bandwidth savings are tantamount. Also, taking advantage of the intrinsics of the query processing engine of the NoSQL store at hand can provide further improvements. We note that MapReduce is a poor match for such complex queries as top-$k$ joins. Our intuition is similar to that of Stonebraker, et al.[23], who pointed out that MapReduce is suboptimal for several aspects of data management, including complex queries. Following this thread of thought, we first overview HRJN[13] and then contribute Inverse Score List rank join (ISL).

### 4.2.1 HRJN Overview

Assume an $n$-way rank join between relations $R_1$, $R_2$, ..., $R_n$. In HRJN the tuples of each relation $R_i$ are sorted in lists, ranked according to a scoring attribute $R_i.score$, or by using a scoring function on one or more attribute values. For simplicity, we assume the former scenario, but our solutions are equally applicable in the latter. The score of the $n$-way join result tuples is then computed using a monotonic ranking function $f(R_1.score, \ldots, R_n.score)$ on the individual scores of joined tuples. Tuples from the $n$ lists are iteratively retrieved in decreasing score order, and the algorithm keeps the minimum ($\bar{s}_i$) and maximum ($\hat{s}_i$) tuple scores seen thus far (for $i \in [1, \ldots, n]$). Every retrieved tuple is joined against previously retrieved ones and appended to the result set, if the latter has less than $k$ tuples or the score of the new join tuple is higher than that of the $k^{th}$ tuple (resulting in the latter's elimination). Then, a threshold score $S$ is computed as: $S = max\{f(\bar{s}_1, \hat{s}_2, \ldots, \hat{s}_n), f(\hat{s}_1, \bar{s}_2, \ldots, \hat{s}_n), \ldots f(\hat{s}_1, \hat{s}_2, \ldots, \bar{s}_n)\}$. In other words, the threshold score equals the maximum attainable score by any subsequent join result tuple. The algorithm terminates when the score of the $k^{th}$ join result is greater than the threshold.

### 4.2.2 The ISL Index

Like HRJN, ISL is based on the existence of inverted score lists. These lists are part of the ISL index, created via a map-only MapReduce job (Algorithm 3), just like in the case of the IJLMR index above. More specifically, for each input relation, we build and maintain an index, comprised of a column family in a common index table, where each row has a score value as its key, and the set of input tuples with this value in their score attribute as the content of the row. A kink of HBase is that it provides fast scans in increasing rowkey order but has no support for scans in the other direction; due to this, in our implementation we have used the negated score values as the index keys (see Fig. 3).

### 4.2.3 ISL Query Processing

The query processing algorithm is outlined in Algorithm 4. During query processing, the "coordinator" scans through the index column families for the joined relations alternately. Scanning is performed in increasing key (i.e., decreasing score) order, and in batches of a user-defined size. As NoSQL stores are in essence column stores, and key-value pairs with subsequent keys are stored next to each-other on disk, batching reads results in a lower disk I/O overhead, as well as a lower processing time due to the cost of IPC calls to the NoSQL store being amortized over the batch size. As we shall see in the performance evaluation section, such batched scans (e.g., HBase scans with a non-zero rowcache size) can result in significant gains in query processing times, trading off bandwidth consumption and dollar-costs. The coordinator stores (in-memory) all retrieved tuples in separate hash tables, using the join value as the key; this allows for fast

**Algorithm 4** ISL Rank-Join

```
1  Input: Rows from the ISL index for column families A
       and B, of the form {row.score:  row.rowKey,
       row.joinvalue} (see Fig. 3), batch sizes C_A, C_B
2  Output: Top-k join result set
3  HashTable tuplesA=∅, tuplesB=∅;
4  SortedList results=∅, batch=∅;
5  CurrentRelation cr = A;
6  while (true) {
7    if (cr == A) {
8      myTuples = tuplesA.get(kv.joinValue);
9      otherTuples = tuplesB.get(kv.joinValue);
10   } else {
11     myTuples = tuplesB.get(kv.joinValue);
12     otherTuples = tuplesA.get(kv.joinValue);
13   }
14   batch.insert(next C_cr rows from CF "cr");
15   foreach(row ∈ batch) {
16     foreach(kv ∈ row) {
17       myTuples.append(kv.joinValue, kv);
18       foreach (kv' ∈ otherTuples) {
19         results.add(innerJoin(kv, kv'));
20         if (HRJNTerminationTest(results,
               tuplesA, tuplesB) == true)
21           return (results);
22       }
23     }
24   }
25   batch.clear();
26   cr = (cr == A) ? B : A;
27 }
28 return (results);
```



**Figure 4: BFHM bucket structure**

joins whenever new tuples are fetched. The coordinator further maintains a list of the current top-$k$ results. With every new tuple fetched and processed, the coordinator computes the current threshold value, and terminates when it is below the score of the $k$'th tuple in the result set.

## 5. STATISTICAL RANK-JOINS

Both of the previous algorithms, ship tuples even though they may not participate in the top-$k$ result set. Our next contribution aims to avoid this. Note that we need not only estimate which tuples will produce the join result, but also to predict whether these tuples can have a top-$k$ score.

### 5.1 The BFHM Data Structure

The BFHM index is a two-level statistical data structure, encompassing histograms and Bloom filters. At the first level, we have an equi-width histogram on the score axis; that is, all histogram buckets have the same spread and each such bucket stores information for tuples whose scores lie within the boundaries of the bucket. At the second level, instead of a simple counter per bucket (plus the actual min and max scores of tuples recorded in the bucket), we choose to maintain a Bloom filter-like data structure, recording the join values of the tuples belonging to the bucket. This will then be used to estimate the cardinality of the join result set during query processing, to be discussed shortly. In brief, the BFHM data structure has two main parameters: the number of buckets in the BFHM ($numBuckets$), and the number of bits in each BFHM bucket Bloom filter ($m$).

As false positives can inflate the join cardinality estimation, we have opted for a fusion scheme, combining single-hash-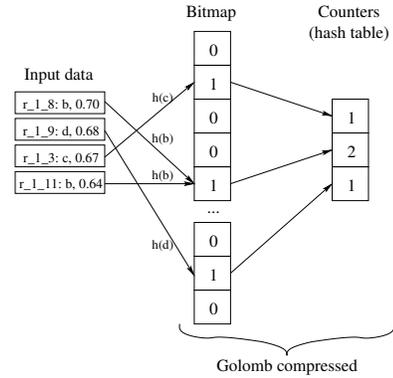function Bloom filters with Counting Bloom filters and compression. More specifically, in each BFHM bucket we maintain: (i) the minimum and maximum score values of tuples recorded in the bucket; (ii) a single-hash-function Bloom filter of size $m$ (bits); and (iii) a hash table of counters for each non-zero bit of the Bloom filter. Both of the latter two constructs are then compressed using Golomb coding[11]. The resulting data structure is a hybrid between Golomb Compressed Sets and Counting Bloom filters, allowing us at the same time to (i) minimize the false positive probability for our treatment of Bloom filters for join cardinality estimation (to be discussed shortly), (ii) greatly reduce the amount of bytes stored in the NoSQL store and transferred across the network, and (iii) achieve a reasonable trade-off between compression ratio and processing costs. Fig. 4 depicts a pictorial of how data are inserted to the Bloom filter-related section of the BFHM bucket for the score range $(0.60, 0.70]$ for tuples of relation $R_1$ in our running example. Please note that the compression of the bit vector and counter hash table is an integral part of our data structure, as single hash function Bloom filters can grow very large in space and are thus impractical otherwise. Moreover, to our knowledge, this is the first work to propose, implement, and evaluate such a fusion scheme.

Like with our indices, the BFHM data are stored in the NoSQL store. More specifically, for each input relation, the BFHM index is stored in a separate column family or index table. Each BFHM bucket is stored in a separate row with the bucket number as its key (e.g., for scores in $[0, 1]$ and 10 buckets, the first bucket – i.e., for score values in $(0.9, 1.0]$ – will be stored under key 0, the bucket for score values in $(0.8, 0.9]$ will use key 1, and so on); the row values then include the min and max actual scores, plus the Golomb-compressed bitmap and counters' hashtable (coined BFHM bucket "blob"). Moreover, as we shall see shortly, during query processing we need to be able to map BFHM set bit positions back to the corresponding join values. As this is not possible with most quasi-random hash functions, we need to further store these mappings. These are stored in the same column family/table as the above data, in rows where the key consists of the concatenation of the bucket number and bit position, and where the row data includes tuples of the form $\{rowkey : join\ value, score\}$. Assuming that $h(x)$ is the bit position indicated for item $x$ by the hash function used by our Bloom filter, Fig. 5 depicts the BFHM table contents for our running example. This is created with a MapReduce job (Algorithm 5). In the Map phase, the map-

**Algorithm 5** BFHM Index Creation

```
1 Input: Rows from a single column family (e.g., A), of
       the form {row.rowKey: row.joinValue, row.score},
       number of buckets in BFHM (numBuckets)
2 Output: BFHM blob rows and reverse mappings for A
3 Map():
4    foreach(row ∈ sequential scan of A) {
5      int bucketNo = scoreToBucket(row.score,
           numBuckets);
6      emit(bucketNo: {row.rowKey:
           row.joinvalue, row.score});
7    }
8 Reduce():
9    HybridBloomFilter filter; // see sec. 5.1
10   float minScore = ∞, maxScore = -∞;
11   foreach(kv ∈ input) {
12     int bitPos = filter.insert(row.joinValue);
13     if (row.score < minScore)
14       minScore = row.score;
15     if (row.score > maxScore)
16       maxScore = row.score;
17     emit(bucketNo|bitPos: {row.rowKey:
           row.joinvalue, row.score});
18   }
19   emit(bucketNo, {GolombCompress(filter),
           minScore, maxScore});
```

| Row key | Index tuples ([blob],score$_{min}$,score$_{max}$ or {row key: join value, score}) | |
|---|---|---|
| | $R_1$ | $R_2$ |
| 0 | [blob],0.93,1.00 | [blob],0.91,0.92 |
| 0\|h(a) | {r1$_{10}$: a,1.00} | |
| 0\|h(b) | | {r2$_2$: b,0.91},{r2$_{11}$: b,0.92} |
| 0\|h(c) | {r1$_2$: c,0.93} | |
| 1 | [blob],0.82,0.82 | |
| 1\|h(b) | {r1$_7$: b,0.82} | |
| 1\|h(d) | {r1$_1$: d,0.82}, {r1$_4$: d,0.82} | |
| 2 | [blob],0.70,0.79 | |
| 2\|h(a) | {r1$_5$: a,0.73} | |
| 2\|h(b) | {r1$_8$: b,0.70} | |
| 2\|h(c) | {r1$_6$: c,0.79} | |
| 3 | [blob],0.64,0.68 | [blob],0.64,0.64 |
| 3\|h(b) | {r1$_{11}$: b,0.64} | |
| 3\|h(c) | {r1$_3$: c,0.67} | {r2$_3$: c,0.64} |
| 3\|h(d) | {r1$_9$: d,0.68} | |
| 4 | | [blob],0.50,0.53 |
| 4\|h(a) | | {r2$_1$: a,0.51} |
| 4\|h(d) | | {r2$_4$: d,0.53}, {r2$_6$: d,0.50} |
| 5 | | [blob],0.41,0.41 |
| 5\|h(d) | | {r2$_5$: d,0.41} |
| 6 | | [blob],0.31,0.38 |
| 6\|h(a) | | {r2$_7$: a,0.35}, {r2$_8$: a,0.38}, {r2$_9$: a,0.37} |
| 6\|h(c) | | {r2$_{10}$: c,0.31} |

**Figure 5: Running example: BFHM index table**

pers partition incoming tuples into the various histogram buckets. Each reducer operates on the mapped tuples for one BFHM bucket at a time. Each incoming tuple is first added to the BFHM hybrid filter based on its join value, and its corresponding bit position is recorded. The reducer emits a reverse mapping entry for each such tuple, and keeps track of the min and max scores of all tuples in the bucket. When the bucket tuples are exhausted, the reducer finally emits the BFHM bucket blob row.

## 5.2 BFHM Query Processing

Query processing consists of two phases: (i) estimating the result, and (ii) reverse mapping and computation of the true result. Algorithm 6 shows the 1st phase. The "coordinator" fetches BFHM bucket rows for the joined relations, one at a time, with newly fetched buckets being "joined" with older ones. The bucket join result – an estimation of the join result for tuples recorded in the joined buckets – is then added to the list of estimated results. When the estimated number of result tuples in this list (i.e., the sum of cardinalities of added buckets) is above $k$, the algorithm tests for the BFHM termination condition, to be discussed shortly. If the latter is satisfied, processing continues with the reverse mapping/final result set computation phase.

Algorithm 7 outlines the bucket join procedure. First, we compute the bitwise-AND of the Bloom filter bitmaps from the two buckets; if the resulting bitmap is empty (i.e., all bits are 0), then there are no joining tuples recorded in these two buckets. Otherwise, we compute an estimation of the cardinality of the join, by summing up the products of the counters corresponding to the non-zero bit positions in the result filter. The factor $\alpha$ (line 9) is there to compensate for false positives in the filters; for now, assume $\alpha = 1$. Last, we compute the min and max score of any join result tuple from these buckets, by using the actual min and max scores of the joined buckets as input to the aggregate score function. Fig. 6(c) shows the estimated result set for our running

example, using *sum* as the aggregate scoring function; the join attribute value is shown as $h(\cdot)$ to denote that we refer to non-zero positions in the bitwise-AND of filter bitmaps and not to actual join values, while the Bloom filter counters are given in consolidated form in Fig. 6(a) and 6(b) for clarity. First, the algorithm would fetch the $(0.9, 1.0]$ buckets for $R_1$ and $R_2$; their bucket-join would return *null* as they have no common non-zero bit position. The algorithm would then proceed by fetching bucket $(0.8, 0.9]$ for $R_1$ and joining it to bucket $(0.9, 1.0]$ for $R_2$; the join would return an estimated result containing two tuples (the product of the counters for bit position $h(b)$), with a minimum score of $0.82+0.91 = 1.73$ and a maximum score of $0.82+0.92 = 1.74$. Then it would be $R_2$'s turn, fetching the $(0.6, 0.7]$ bucket and joining it to the two buckets already fetched for $R_1$, etc.

To test for the termination of the estimation phase, we examine the estimated results list and the buckets fetched so far. First, we compute the minimum score of the $k$'th estimated result. The estimation phase terminates if there are more than $k$ estimated results and there is no combination of buckets not examined so far that could have a maximum score above that of the $k$'th estimated result.

Take for example the estimated result of Fig. 6(c) and assume we requested the top-3 join results. After having fetched the first two buckets for $R_1$ (i.e., $(0.9, 1.0]$, $(0.8, 0.9]$) and for $R_2$ (i.e., $(0.9, 1.0]$ and $(0.6, 0.7]$), we would have computed rows 1 and 3 of the result set in Fig. 6(c). At this time, the estimated result set consists of $2 + 1 = 3$ estimated tuples, and the minimum score of the third tuple would be 1.57. The maximum attainable score for the join of the next bucket (i.e., bucket $(0.7, 0.8]$) of $R_1$ and the highest-score bucket of $R_2$ (i.e., $(0.9, 1.0]$) would be $0.8 + 1.0 = 1.8$ which is higher than 1.57 so the estimation phase does not terminate. After fetching and joining bucket $(0.7, 0.8]$ of $R_1$, the result set would consist of rows 1, 2, 3, and 6 of Fig. 6(c). Now the estimated score for the top-third result becomes 1.71. The maximum attainable score for the join of the next bucket of $R_2$ (bucket $(0.5, 0.6]$) against the highest-scoring

**Algorithm 6** BFHM Rank-Join Estimation

```
1  Input: Rows from the BFHM index for A and B (Fig. 5)
2  Output: Estimated top-k join result set
3  List bucketsA=∅, bucketsB=∅, myBuckets,
       otherBuckets;
4  SortedList results=∅; // Sorted on maxScore
5  int numEstimatedResults = 0;
6  CurrentRelation cr = A;
7  BFHM newBucket;
8  boolean done = false;
9  while (!done) {
10   if (cr == A) {
11     myBuckets = bucketsA;
12     otherBuckets = bucketsB;
13   } else {
14     myBuckets = bucketsB;
15     otherBuckets = bucketsA;
16   }
17   newBucket = fetchNextBucketFrom(BFHM_{cr});
18   myBuckets.add(newBucket);
19   foreach(bucket ∈ otherBuckets) {
20     EstimatedResult res = bucketJoin(newBucket,
           bucket); // See alg. 7
21     if (res == null)
22       continue;
23     results.add(res);
24     numEstimatedResults += res.cardinality;
25     if (BFHMTerminationTest(bucketsA,bucketsB,
           numEstimatedResults)) {
26       done = true;
27       break;
28     }
29   }
30   cr = (cr == A) ? B : A;
31 }
32 return (results);
```

**Algorithm 7** BFHM bucket join

```
1  Input: BFHM_{R_1}[i] (i'th bucket from R_1's
       BFHM), BFHM_{R_2}[j] (j'th bucket from R_2's BFHM)
2  Output: join result estimation for this bucket pair
3  EstimatedResult res;
4  res.BF = BFHM_{R_1}[i].BF & BFHM_{R_2}[j].BF;
5  if (res.BF == ∅)
6      return null;
7  foreach(bit ∈ res.BF non-zero bits)
8      res.cardinality += BFHM_{R_1}[i].counters(bit) *
           BFHM_{R_2}[j].counters(bit) * α;
9  res.minScore = joinScore(BFHM_{R_1}[i].minScore,
       BFHM_{R_2}[j].minScore);
10 res.maxScore = joinScore(BFHM_{R_1}[i].maxScore,
       BFHM_{R_2}[j].maxScore);
11 return res;
```

bucket of $R_1$ would be 1.6; conversely, the maximum attainable score for the next bucket of $R_1$ (bucket $(0.6, 0.7]$) against bucket $(0.9, 1.0]$ of $R_2$ is 1.7; since both of these are lower than 1.71, the estimation phase terminates.

The next phase examines the estimated results of the first phase and purges all estimated results whose maximum score is below that of the (estimated) $k$'th tuple. Then, the algorithm fetches the reverse mapping rows corresponding to the non-zero bit positions of the Bloom filters in the estimated results, which are then used to compute the final result set.

## 5.3 Analysis of BFHM Rank-Join

Our BFHM-based algorithms deal with two sources of inaccuracy when estimating the rank join result set: use of histograms and use of Bloom Filters. The former introduces errors in the estimation of the actual score of the join results, while false positives in the latter may result in overestimating the join result size. For ease of presentation, assume for now that our Bloom filters are false-positive free. This allows us to know for sure when joining tuples from any two given buckets of the BFHM will actually produce join results. However, it gives us no way of knowing the exact scores of the joined tuples and thus does not allow us to compute the actual score of the join result tuple.

In order to accomplish this, we maintain the min and max score achievable when joining two buckets; then, instead of keeping the $k$ highest scored estimated results, our algorithms also keep all those tuples whose maximum possible score is larger than the lowest possible score of the $k^{th}$ es-

timated result. This guarantees that no tuple is lost from the final result set, at the expense of fetching/storing some tuples that may not make it in the final result set.

In a false-positive-free world, this would suffice. Alas, false positives in the Bloom filters of the BFHM may cause an overestimation of the join result set size for any two joined buckets. Surely one can tweak the Bloom filter parameters so as to minimize the false positive probability; however, doing this may lead to overly large Bloom filters, thus diminishing any bandwidth consumption returns expected from their use. Moreover, even if very large BFs were practical, one can not guarantee that no false positives arise, as a result for example of a BFHM bucket being overpopulated. In order to deal with this, we incorporate the effective false positive probabilities of the Bloom filters in the join result size estimation (the $\alpha$ factor in algorithm 7). Given a single-hash-function Bloom filter of size $m$, after having inserted $n$ distinct items, the probability that a given bit is set equals $P_T = (1 - (1 - 1/m)^{kn}) \approx 1 - e^{-m/kn}$. In the case of counting Bloom filters, we can use this information to estimate a "compensated" value of any given counter, as follows. When joining two filters (say $BF_A$ and $BF_B$), and $JSize$ is the estimated join size as computed through $BF_A \cdot BF_B$ (i.e., sum of products of matching Bloom filter counter values), we scale $JSize$ by a factor of $\alpha = (1 - P_{T_A}) \cdot (1 - P_{T_b})$; that is:
$$JSize_{A \cdot B} = BF_A \cdot BF_B \cdot (1 - P_{T_A}) \cdot (1 - P_{T_b})$$

Our experimental evaluation showed that the combination of these two mechanisms results in a 100% recall for all workloads and parameter values tested. Apart from the above probabilistic scheme, we can further *guarantee* a 100% recall, as follows. First, when we have $k$ or more results in the final result set, we examine the score of the $k^{th}$ actual join result and compare it to the maximum scores of BFHM buckets that didn't make it to the fetch list. If there are buckets whose maximum score is above the former, then we should consider these additional buckets too. If no change occurs in the result set after this step, the algorithm terminates. Similarly, for the case were $k' < k$ results have been produced in the second query processing phase, we resume the query processing algorithm from the point it initially stopped, only now looking for the top-$k + (k - k')$ results. When $k$ or more results have been produced, the algorithm performs the checks outlined in the first case.

LEMMA 1. *The set of tuples represented by the BFHM resulting by combining the BFHMs of multiple joined buckets,*

| | 0.9–1.0 | 0.8–0.9 | 0.7–0.8 | 0.6–0.7 |
|---|---|---|---|---|
| min | 0.93 | 0.82 | 0.70 | 0.64 |
| max | 1.00 | 0.82 | 0.79 | 0.68 |
| h(a) | 1 | | 1 | |
| h(b) | | 1 | 1 | 1 |
| h(c) | 1 | | 1 | 1 |
| h(d) | | 2 | | 1 |

(a) $R_1$ BFHM

| | 0.9–1.0 | 0.6–0.7 | 0.5–0.6 | 0.4–0.5 | 0.3–0.4 |
|---|---|---|---|---|---|
| min | 0.91 | 0.64 | 0.50 | 0.41 | 0.31 |
| max | 0.92 | 0.64 | 0.53 | 0.41 | 0.38 |
| h(a) | | | 1 | | 3 |
| h(b) | 2 | | | | |
| h(c) | | 1 | | | 1 |
| h(d) | | | | 2 | 1 |

(b) $R_2$ BFHM

| # | Join Attr | Min Score | Max Score | # of est. Results | $R_1$ bucket | $R_2$ bucket |
|---|---|---|---|---|---|---|
| 1 | h(b) | 1.73 | 1.74 | 2 | 0.8–0.9 | 0.9–1.0 |
| 2 | h(b) | 1.61 | 1.71 | 2 | 0.7–0.8 | 0.9–1.0 |
| 3 | h(c) | 1.57 | 1.64 | 1 | 0.9–1.0 | 0.6–0.7 |
| 4 | h(b) | 1.55 | 1.60 | 2 | 0.6–0.7 | 0.9–1.0 |
| 5 | h(a) | 1.43 | 1.53 | 1 | 0.9–1.0 | 0.5–0.6 |
| 6 | h(c) | 1.34 | 1.43 | 1 | 0.7–0.8 | 0.6–0.7 |
| 7 | h(d) | 1.32 | 1.35 | 4 | 0.8–0.9 | 0.5–0.6 |
| 8 | h(c) | 1.28 | 1.32 | 1 | 0.6–0.7 | 0.6–0.7 |
| 9 | h(a) | 1.24 | 1.38 | 3 | 0.9–1.0 | 0.3–0.4 |
| 10 | h(c) | 1.24 | 1.38 | 1 | 0.9–1.0 | 0.3–0.4 |
| 11 | h(d) | 1.23 | 1.23 | 2 | 0.8–0.9 | 0.4–0.5 |
| 12 | h(a) | 1.20 | 1.32 | 1 | 0.7–0.8 | 0.5–0.6 |
| 13 | h(d) | 1.14 | 1.21 | 2 | 0.6–0.7 | 0.5–0.6 |
| 14 | h(d) | 1.05 | 1.09 | 1 | 0.6–0.7 | 0.4–0.5 |
| 15 | h(a) | 1.01 | 1.17 | 3 | 0.7–0.8 | 0.3–0.4 |
| 16 | h(c) | 1.01 | 1.17 | 1 | 0.7–0.8 | 0.3–0.4 |
| 17 | h(c) | 0.95 | 1.06 | 1 | 0.6–0.7 | 0.3–0.4 |

(c) Estimated BFHM join result (score function: *sum*)

**Figure 6: Example: BFHM join result estimation**

*is a superset of the actual join result set.*

PROOF. Remember that tuples are inserted to the BFHM based on their join attribute value, and that when joining two (or more) such structures, we first perform a bitwise-AND of the Bloom filters. This means that, in the resulting BFHM, bit positions that were unset in at least one of the BFHMs, will also be 0, while all remaining positions (i.e., for which all BFHMs had a non-zero value) will be non zero, and the product of the respective counters will give us an estimation of their join result size. Being based on Bloom filters, each individual BFHM cell can only introduce false positives; that is, in our context, the individual counters in every counter position of the original BFHMs will be equal to or larger than the cardinality of the values they represent. Hence, we can only overestimate the number of join results corresponding to any position in the final BFHM. □

In essence, this means that the recall of our BFHM-based algorithm is not affected by the use of Bloom filters. It thus suffices to prove that our treatment of BFHM cells/buckets is such that if some item is missing from the output, then our algorithms can detect and fetch it.

THEOREM 1. *The BFHM-based rank join algorithms can achieve a 100% recall for any valid input.*

PROOF. We shall prove this by contradiction. Let $t$ be a join result tuple which should be in the top-$k$ join result set but is omitted by our algorithm; that is, $t$'s score is among the actual top-$k$ join result scores but $t$ is not in the final result set computed during phase 2 (and possible repetitions, as discussed above). Given lemma 1, this may happen only if the algorithm has stopped before examining the join result BFHM bucket in which $t$ belongs. This in turn means that the result set consists of at least $k$ results,

and that the maximum score of $t$'s bucket (being larger or equal to $t$'s score) – and hence $t$'s score – is below the score of the $k^{th}$ result; a contradiction. □

## 6. UPDATES AND MAINTENANCE

Both the IJLMR and ISL indexes are in essence space-optimized inverted lists of the base data. To maintain these indexes up-to-date in the face of concurrent run-time updates, we have overloaded the base data insertion and deletion functions, intercepting these primitives so as to also propagate changes to the index. More specifically, both insertions and deletions are intercepted at the caller level; then, the mutation is augmented so as to perform both a base data *and* an index insertion/deletion in one operation, using the original mutation timestamp for both operations. This reduces the time between data and index updates, and takes a step towards index consistency. We have opted for eventual consistency, since this is the consistency level also natively supported by most contemporary NoSQL stores; failed mutations are retried until successful and key-value timestamps are used to discern between fresh and stale tuples.

For BFHM, the existence of the BFHM blobs makes concurrent updates more complicated. To this end, we have taken a hybrid approach, where updates to the reverse mappings are performed just as above, while updates to the blobs are handled through special *insertion* and *tombstone* records. These are key-value pairs that are stored in the bucket row, along with the blob and bucket score range. Each tuple insertion in a specific BFHM bucket will result in an "insertion" record being added to the bucket row (in addition to an entry being added in the corresponding reverse mapping row); this key-value pair holds all BFHM-related information (i.e., the tuple's rowkey, join value, and score), and bears the same timestamp as the newly inserted tuple. Conversely, if a tuple is deleted from a BFHM bucket, then a "tombstone" record is added to the bucket row, again bearing all BFHM-related information and the same timestamp as the delete operation; reverse mappings are directly deleted, using the NoSQL store's vanilla delete operation. This information allows anyone retrieving a bucket row to replay all row mutations in timestamp order and reconstruct the up-to-date blob from the original blob. The blob is then written back to the NoSQL store using the timestamp of the latest replayed mutation, and insertion/tombstone records with an older or equal timestamp are purged, all in a single operation. HBase (and most NoSQL stores) support row-level atomicity; coupled with the above treatment of timestamps, this ensures that no updates are lost. The blob write-back can be performed *eagerly* (at the beginning of query processing), *lazily* (after the query results are returned to the user), or *off-line* (by a thread periodically probing bucket rows for mutation records). Moreover, one can choose to perform the write-back only if the number of replayed mutations is above some predefined threshold.

## 7. EXPERIMENTAL EVALUATION

### 7.1 Methodology

We implemented all of the above mentioned algorithms, comprising approx. 6k lines of Java code. We further implemented the DRJN algorithm from [8]. The DRJN index

is roughly a 2-d matrix, with join value partitions on its x-axis and score value partitions on its y-axis. DRJN query processing proceeds as follows: (i) the querying node fetches complete DRJN matrix rows in decreasing score order; (ii) the relevant buckets are "joined" so as to estimate the cardinality of their join; (iii) when the cumulative cardinality surpasses $k$, contact all nodes and fetch and join all tuples whose score is above the the lower score boundaries of the last fetched buckets; (iv) terminate if the cardinality of the actual result set is $k$ and the score of the $k$'th tuple is larger than the maximum attainable join score of the last fetched buckets, otherwise loop to (i), incrementally fetching more buckets and tuples. As [8] was designed for a P2P-like setting, we had to revisit it so as to be useable in a NoSQL store such as HBase. First, we opted to group DRJN buckets by their scores and store all buckets for a given score range as columns of a single row; thus, the querying node will retrieve a complete batch of buckets, as required at step (i) above, with a single HBase *Get()* operation. We further augmented HBase with custom server-side filters to allow for efficient filtering of tuples in step (iv). Last, we further expedited step (iv) by implementing it as a lightweight Map-only Hadoop job, storing its output data in a temporary HBase table for the querying node to access and join.

We employed two different clusters: one in a controlled lab environment and one "in-the-wild". For the latter, we used Amazon's Elastic Compute Cloud (EC2), with clusters consisting of 3, 5, and 9 m1.large nodes. (each with 2 virtual cores, 7.5 GB RAM, and 2x 420 GB of instance storage). The lab cluster (LC) consisted of 5 nodes, each with 2 CPUs, 16 cores per CPU, 64GB RAM, and 10× 1TB disks.

We used the TPC-H generator, generating data for the "Lineitem", "Orders", and "Part" tables, for scale factors from 10 to 500. The larger (smaller) data scale resulted in tables with 3 billion (60M), 750 million (15M), and 100 million (2M) rows, which occupied ≈1.7TB (34GB), ≈200GB (4GB), and ≈25GB (0.5GB) of HBase disk space, respectively. With the TPC-H generator we also computed update sets, to be applied to the base data and indexes. All Bloom filters were configured to contain the most heavily populated of the buckets with a false positive probability of 5%. The number of BFHM buckets was set to 100 and 1000 on EC2 and to 100 and 500 on LC. ISL was configured with batching sizes matching the number of BFHM buckets; 1% and 0.1% on EC2, and 1% and 0.2% on LC. DRJN was also configured with 100 and 500 buckets on LC.

We used the following queries:

Q1: SELECT * FROM Part P, Lineitem L
WHERE P.PartKey=L.PartKey
ORDER BY (P.RetailPrice * L.ExtendedPrice)
STOP AFTER k

Q2: SELECT * FROM Orders O, Lineitem L
WHERE O.OrderKey=L.OrderKey
ORDER BY (O.TotalPrice + L.ExtendedPrice)
STOP AFTER k

These queries were selected to showcase both the use of different aggregate scoring functions and the effect of score value distributions on the query processing time. Queries were executed 20 times and we report on the average values (the standard deviation was too small to show on the graphs in all cases). Last, we applied the update sets one at a time and executed 20 repetitions of the same queries. All block-level/memory caches were purged between consecutive

update and query executions.

We evaluate all algorithms using the following metrics:

- Turnaround time: the wall-clock time required to compute and return the top-$k$ join result.
- Network bandwidth: the number of bytes transferred through the network.
- Dollar cost: the number of tuples read from the cloud store during query processing[1].

Regarding query processing times, the BFHM and ISL algorithms were similar across all EC2 cluster sizes. For the PIG, HIVE, and IJLMR approaches, the increase in cluster size resulted in a ≈30% decrease in processing time going from 1+2 to 1+8 nodes, with the rest of the metrics being roughly the same across cluster sizes. Thus, to save space, we present only the figures for the 1+8 EC2 cluster and a scalefactor of 10 (denoted "EC2"), and for the 5-node lab cluster and a scalefactor of 500 ("LC"). With respect to query response time, IJLMR, PIG, and HIVE had significantly reduced performance. Specifically, IJLMR, was consistently worse than the next-best algorithm by up to an order of magnitude, PIG was worse than IJMLR by about an order of magnitude, and HIVE was worse than PIG by about an order of magnitude. Thus, for presentation clarity we omit specific results for these when showing the LC results with the big scale factor.

## 7.2 Results

*Query Processing Time.* Figures 7(a), 7(d), 8(a), and 8(d) depict the time required by each algorithm to process the Q1 and Q2 top-$k$ join queries, for various values of $k$. Please note the logarithmic y-axis. Contrasting the results for Q1 and Q2 we can see how the different score distributions affect the processing time. For Q2 there are fewer high-ranking tuples, thus we need to reach deeper into each index to produce the top-$k$ result set compared to Q1. On EC2, BFHM is the clear winner across the board, with ISL following, and IJLMR, PIG, and HIVE trailing by large margins. For LC, ISL is shown to be best, with BFHM closing the gap and occasionally beating ISL, as $k$ increases. DRJN trails by several orders of magnitude, primarily due to the cost of the Map jobs needing to scan the whole dataset to send to the coordinator those rows having a score greater that the threshold calculated by it.

*Query Processing Bandwidth Consumption.* Figures 7(b), 7(e), 8(b), and 8(e) depict the bandwidth consumed to process Q1 and Q2. IJLMR does in general very well, as it only transfers the local top-$k$ lists from the mappers to the sole reducer. However, as $k$ increases, BFHM closes the gap, eventually even winning for large values of $k$. In general, DRJN achieves its best performance for this metric. From the LC results, DRJN is the clear winner for Q1 and for low-$k$ values for Q2. Note that in DRJN, although its mappers need to scan the complete dataset, this is typically fetched from each mapper's local disk. Further, our optimization of server-side filtering paid off, as the amount of data put on

---

[1]Per DynamoDB's pricing scheme[9], each key-value read from the NoSQL store corresponds to 1 unit of *Read Capacity* (as all of our key-value pairs are less than 1KB in size), with *Read Throughput* being priced at \$0.01 per hour for every 50 units of Read Capacity.
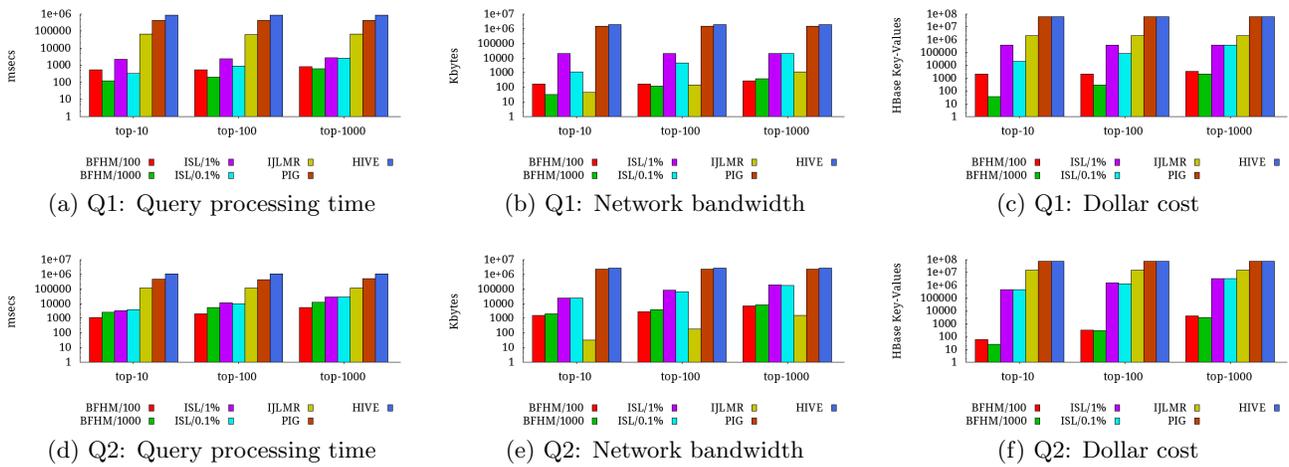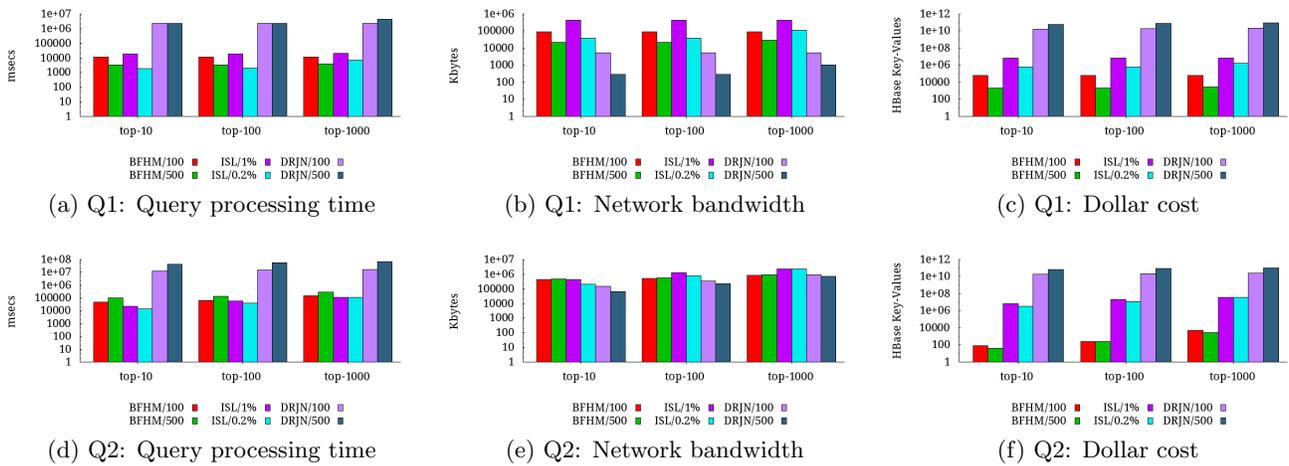
(a) Q1: Query processing time
(b) Q1: Network bandwidth
(c) Q1: Dollar cost

(d) Q2: Query processing time
(e) Q2: Network bandwidth
(f) Q2: Dollar cost

**Figure 7: Results for Q1 and Q2 on EC2**



(a) Q1: Query processing time
(b) Q1: Network bandwidth
(c) Q1: Dollar cost

(d) Q2: Query processing time
(e) Q2: Network bandwidth
(f) Q2: Dollar cost

**Figure 8: Results for Q1 and Q2 on LC**

the network is significantly reduced. For Q1, where the top-k join is computed very early, DRJN shines, as very little data need be fetched over the network. For the more demanding Q2 however, as $k$ increases, its improvement over BFHM becomes much smaller.

*Query Processing Dollar-cost.* Following the DynamoDB pricing model, Figures 7(c), 7(f), 8(c), and 8(f) depict the number of key-value pairs read from the NoSQL store. Naturally, the MapReduce approaches are the worst, since they need to scan all of the input data. BFHM, with its accuracy in estimating the result set cardinality, and its "surgical" accuracy in retrieving appropriate tuples from the input relation, is the clear winner here with ≈1-3 orders of magnitude less cost than the next best contender (ISL) and up to 5 orders of magnitude better than DRJN.

*Indexing Costs.* Fig. 9 depicts the indexing times, showing that our indexing algorithms scale well with the cluster and dataset sizes. We stress that, across the board, the sum of the index building time plus the relevant query processing times shown earlier, is on par or lower than the time required

to execute the same query in PIG (and much faster than executing it in HIVE). In essence, this means that we can afford to build our indices just before executing a query, and still be competitive against PIG or HIVE! Additionally, we report on the storage space used by each index and the maximum memory footprint of individual mappers/reducers during the index building stages, on the Lab Cluster and for the 500-scalefactor. More specifically, the disk space used by each index (for Part, Orders, and Lineitem resp.) was:

- BFHM: 2.6, 22, and 110 GB (incl. reverse mappings)
- ISL: 1.2, 13.5, and 85 GB
- IJLMR: 1.2, 13.5, and 85 GB
- DRJN: ranging from 400 kB (100 buckets) to 8.5 MB (500 buckets)

Keep in mind that the on-disk size of the base relations was 25 GB, 200 GB, and 1.7 TB respectively for Part, Orders, and Lineitem. The memory footprint of reducers during the index building phase was:

- BFHM/100 buckets: 4 GB worst case, 1 GB average.
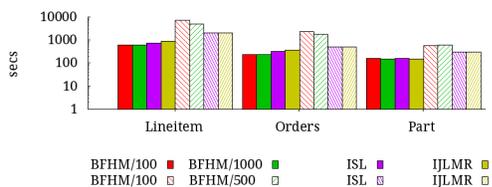- BFHM/500 buckets: 2 GB worst case, 0.5 GB average
- ISL/IJLMR: negligible

**Figure 9: Indexing time (solid: EC2; pattern: LC)**

- DRJN: ranging from 3.5 MB (Part, 500 buckets) to 125 MB (Lineitem, 100 buckets)

***Online Updates.*** Last, we studied the effect of online updates for BFHM. We first used the TPC-H generator to generate a number of update sets, each consisting of $\approx s \times 600$ insertions and $\approx s \times 150$ deletions for scale-factor $s$. We then applied each of these sets in their entirety (i.e., $\approx 750$ mutations), followed by a single query for which we measured the query processing time. Even with the "eager" update scheme (i.e., the coordinator reconstructed and wrote back the updated BFHM at the beginning of query processing), fitting an update-heavy workload – a worst-case scenario with regard to the query processing time overhead – the overall time overhead was less than 10% across the board (figure omitted due to space reasons).

## 8. CONCLUSIONS

Top-k join queries arise naturally in many real-world settings. We studied algorithms for rank joins in NoSQL stores. This is, to our knowledge the first such endeavor. We contributed novel algorithms, implemented them, and extensively tested their performance over Amazon EC2 and in-house clusters, using TPC-H data at various scales and different query types. The central conclusion is that for all metrics, data sets, and query types studied, the BFHM Rank Join algorithm is very desirable. It typically manages to outperform the others and even when it is not the best approach, it offers a performance that is in absolute terms satisfactory and is less sensitive to the various query types, data sets, $k$ values, and their configuration parameters. Immediate future plans include the adoption of dynamic Bloom filters to further improve the time and bandwidth performance of BFHM Rank Join, as well as an exploration of the design space with regard to our various system parameters.

## 9. REFERENCES

[1] A. Abouzeid, et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proc. EDBT*, 2010.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[4] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proc. ICDE*, 2001.

[5] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. ACM PODC*, 2004.

[6] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proc. ACM SIGMOD*, 2003.

[7] J. Dittrich, et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.

[8] C. Doulkeridis, et al. Processing of rank joins in highly distributed systems. In *IEEE ICDE*, 2012.

[9] DynamoDB pricing scheme: `http://aws.amazon.com/dynamodb/#pricing`.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM PODS*, 2001.

[11] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399, 1966.

[12] I. Ilyas, W. Aref, and A. Elmagarmid. Joining ranked inputs in practice. In *Proc. VLDB*, 2002.

[13] I. Ilyas, W. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. In *Proc. VLDB*, 2003.

[14] I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):1–58, 2008.

[15] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proc. ACM SIGMOD*, 2011.

[16] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proc. VLDB*, 2005.

[17] M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[18] J. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.

[19] A. Natsev, Y.-C. Chang, J. Smith, C.-S. Li, and J. Vitter. Supporting incremental join queries on ranked inputs. In *Proc. VLDB*, 2001.

[20] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proc. ACM SIGMOD*, 2011.

[21] C. Olston, et al. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[22] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *Proc. ACM PODS*, 2008.

[23] M. Stonebraker, et al. Mapreduce and parallel DBMSs: Friends or foes? *Comm. ACM*, 53(1):64–71, 2010.

[24] A. Thusoo, et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[25] M. Wu, L. Berti-Equille, A. Marian, C. Procopiuc, and D. Srivastava. Processing top-k join queries. *PVLDB*, 3(1-2):860–870, 2010.

[26] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for kNN join processing. In *Proc. VLDB*, 2004.

[27] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *Proc. ICDE*, 2009.

[28] D. Zeinalipour-Yazti, et al. The Threshold Join Algorithm for top-k queries in distributed sensor networks. In *Proc. ACM DMSN*, 2005.

[29] K. Zhao, S. Zhou, K.-L. Tan, and A. Zhou. Supporting ranked join in peer-to-peer networks. In *Proc. DEXA*, 2005.