Industrial and Applications Paper

Supporting Similarity Queries in Apache AsterixDB

Taewoo Kim[1] Wenhai Li[2] Alexander Behm[†] Inci Cetindil[†] Rares Vernica[†]
Vinayak Borkar[†] Michael J. Carey[1] Chen Li[1]

[1]University of California, Irvine, CA, USA [2]Wuhan University, China

ABSTRACT

Many applications require similarity query processing. Most existing work took an algorithmic approach, developing indexing structures, algorithms, and/or various optimizations. In this work, we choose to take a different, systems-oriented approach. We describe the support for similarity queries in Apache AsterixDB, a parallel, open-source Big Data management system for NoSQL data. We describe the lifecycle of a similarity query in the system, including the support provided at the query language level, indexing, execution plans (with and without indexes), plan rewrites to optimize query execution, and so on. Our approach leverages the existing infrastructure of AsterixDB, including its operators, parallel query engine, and rule-based query optimizer. We have conducted an experimental study using several large, real data sets on a parallel computing cluster to evaluate AsterixDB's support for similarity queries, and we share the efficacy and performance results here.

1 INTRODUCTION

Similarity queries compute answers satisfying matching conditions that are not exact but approximate. The problem of supporting similarity queries has become increasingly important in many applications, including search, record linkage [1], data cleaning [27], and social media analysis [4]. For instance, during a live phone conversation with a client, a call center representative might wish to immediately identify a product purchased by the customer by typing in a serial number. The system should locate the product even in the presence of typos in the search number. A social media analyst might want to find user accounts that share common hobbies or social friends. A medical researcher may want to search for papers whose title is similar to a particular article. In each of these examples the query includes a matching condition with a similarity function that is domain specific, such as edit distance for a keyword or Jaccard for sets of hobbies.

There are two basic types of similarity queries. One is *search*, or *selection*, which finds records similar to a given record. The other is *join*, which computes pairs of records that are similar to each other. There have been many studies on these two types of queries, both with and without indexes. A plethora of data structures, partitioning schemes, and algorithms have been developed to support similarity queries efficiently on large data sets. When the computation is beyond the limit of a single computer, there are also parallel solutions that support queries across multiple nodes in a cluster. (See Section 1.1 for an overview.) The techniques developed in the last two decades have significantly improved the performance of similarity queries and have enabled applications to support such queries on millions or even billions of records.

Most existing work has taken an algorithmic approach, developing index structures and/or algorithmic optimizations. We have taken a different, systems-oriented approach – tackling the problem of supporting similarity queries *end-to-end* in a full, declarative parallel data management system setting. Here we explain how such queries are supported in Apache AsterixDB, an open-source parallel data management system for semi-structured (NoSQL) data. By "end-to-end", we refer to the whole lifecycle of a query, including query language support for similarity conditions, internal index structures, execution plans with or without an index, plan rewriting to optimize execution, and so on.

Achieving our goal has involved several challenges. First, as similarity in queries can be domain specific, we need to support commonly used functions as well as letting users provide their own customized functions. Second, due to the complex logic of existing algorithms, we need to consider how to support them using existing operators without "reinventing the wheel" (without introducing new, ad hoc operators). Third, we need to consider how to leverage an existing query optimizer framework to rewrite similarity queries to achieve high performance. In this paper we discuss these challenges and offer the following contributions:

(1) We show how to extend the existing query language of AsterixDB to allow users to specify a similarity query, either by using a system-provided function or specifying their own logic as a user-defined function (Section 3).

(2) We show how to implement state-of-the-art techniques using existing operators in AsterixDB, both for index-based and non-index-based plans (Section 4) and for both search and join queries. Our solution not only allows the query plans to benefit from the built-in optimizations in those operators, but also to automatically enjoy future improvements in these operators.

(3) We show how to rewrite similarity queries in an existing rule-based optimization framework (Section 5). A plan for an ad hoc similarity join can be very complex. As an example, a three-stage join plan based on the technique in [34] can involve up to 77 operators (Section 5.2). To enable the optimizer to more easily transform such complex plans, we developed a novel framework called "AQL+" that takes a two-step approach to rewriting a plan. A major advantage of the framework is that it allows AsterixDB to support queries with more than one similarity join condition, making it the first parallel data management system (to our knowledge) to support similarity queries with multiple similarity joins.

(4) We present an empirical study using several large, real data sets on a parallel cluster to evaluate the effects of these techniques. The results show the efficacy of AsterixDB's support for parallel similarity queries on large data sets. (Section 6).

1.1 Related Work

There are various kinds of similarity queries on strings and sets. For string similarity search, many algorithms use a gram-based approach (e.g., [5, 20, 29]). VGRAM [19] extends the approach by introducing variable-length grams. For string similarity join, filtering techniques are widely used. Length filtering uses the

[2]Contact author and part of his work was done when visiting UC Irvine. [†]Work done when affiliated with UC Irvine.

length of a string to reduce the number of candidates. For example, an algorithm called gram-count [15] utilizes the fact that for two strings to be similar based on a threshold δ, their length difference should be within δ. Prefix filtering [3, 7, 12, 22, 26, 28, 35, 37–39] utilizes the fact that two strings are similar only if they share some commonality in their prefixes. Many algorithms have been proposed based on this observation, such as All-Pair [3], PPJoin [39], PPJoin+ [39], MPJoin [28], ED-Join [38], AdaptJoin [35], QChunk [26], VChunk [37], and Pivotal prefix [12]. Other related algorithms exist such as M-Tree [9] and trie-Join [14]. There have been several evaluation studies about string-similarity [18] and set-similarity joins [24]. There is a recent survey about string similarity queries [25]. The authors of [18] found that AdaptJoin [35] and PPJoin+ [39] were best for Jaccard similarity. Meanwhile, the authors of [24] concluded that AllPair [3] was still competitive. The authors of [25] discussed prefix-filtering techniques. Many of these algorithms assume the data to be searched or joined can fit in main memory.

For parallel similarity join, a number of studies have used the MapReduce framework [11, 21, 31, 34, 36]. There is one survey that discusses parallel similarity join [13]. Vernica et al. [34] proposed a three-stage algorithm in such a setting. There are also studies on integrating similarity join into database management systems [10, 15, 16, 30, 32]. Some adopted similarity join as a UDF or express a similarity join in a SQL expression; others introduced a relational operator to support similarity joins.

Our focus is different, as it is about supporting similarity in a general-purpose parallel database system context. We needed to address various systems-oriented challenges when adopting existing techniques in this context. System-wise, a parallel similarity query processing system, Dima [33], has been proposed recently. A key difference is that Dima is an in-memory based system, unlike AsterixDB. There are some search systems and DBMSs that support similarity queries, including Elasticsearch, Oracle, and Couchbase. Unlike AsterixDB, Elasticsearch is middleware and it focuses on search, not join. Oracle supports edit distance via an extension package if a specific type of index is created. Couchbase supports edit distance searches on NoSQL data in its new full-text search service, but only via a separate full-text API (not its N1QL query language). In contrast, AsterixDB provides a general class of similarity functions for strings that work for both select and join operations, and a similarity predicate can be part of a general query along with non-similarity predicates.

2 PRELIMINARIES

2.1 Similarity Functions

A similarity measure is used to represent the degree of similarity between two objects. An object can be a string or a bag of elements. There are various types of similarity measures depending on the objects that are being compared. In this paper, we focus on two widely used classes of measures, namely string-similarity functions and set-similarity functions.

String-Similarity Functions: One widely used string similarity function is edit distance, also known as Levenshtein distance. The edit distance between two strings r and s is the minimum number of single-character operations (insertion, deletion, and substitution) required to transform r to s. For instance, the edit distance between "`james`" and "`jamie`" is 2, because the former can be transformed to the latter by inserting "`i`" after "`m`" and deleting "`s`". There are other string-similarity functions such as Hamming distance and Jaro-winkler distance.

Set-Similarity Functions: These are used to represent the similarity between two sets. There are many such functions, such as Jaccard, dice, and cosine. In this paper, we focus on Jaccard similarity, which is one of the most common set-similarity measures. For two sets r and s, their Jaccard similarity is $Jaccard(r,s) = \frac{|r \cap s|}{|r \cup s|}$. For example, the Jaccard similarity between r = {"`Good`", "`Product`", "`Value`"} and s = {"`Nice`", "`Product`"} is $\frac{1}{4}$. Such set-similarity functions can also be utilized to measure the similarity between two strings by tokenizing them (i.e., into n-grams or words) and measuring the set similarity of their token multisets. Dice and cosine values can be calculated similarly.

Similarity Search: Similarity search finds all objects in a collection that are similar to a given object based on a given similarity metric. Let sim be a similarity function, and δ be a similarity threshold. An object r from a collection R is similar to a query object q if $sim(r,q) \geq \delta$.

Similarity Join: Joins find similar $\langle r,s \rangle$ pairs of objects from two collections R and S, where $r \in R$, $s \in S$, and $sim(r,s) \geq \delta$.

2.2 Answering Similarity Queries

For similarity queries, using a brute-force, scan-based algorithm is computationally expensive, so there have been many studies in the literature to support similarity queries more efficiently. One widely used method is the gram-based approach, which utilizes the n-grams of a string. An n-gram of a string r is a substring of r with length n. For instance, the 2-grams of string "`james`" are {"`ja`", "`am`", "`me`", "`es`"}.

review-id	username	review-summary
1	james	This movie touched my heart!
2	mary	The best car charger I ever
3	mario	Different than my usual but good
4	jamie	Great Product - Fantastic Gift
5	maria	Better ever than I expected

Figure 1: Example data of Amazon reviews (simplified).

String-similarity queries can be answered by utilizing an n-gram inverted index. For each gram g of the strings in a collection R, there is an inverted list l_g of the ids of the strings that include this gram g. Figure 2 shows the inverted lists for the 2-grams of the "username" field of the sample Amazon reviews in Figure 1.

gram	am	ar	es	ia	ie	io	ja	ma	me	mi	ri	ry
inverted list	1 4	2 3 5	1	5	4	3	1 4	2 3 5	1	4	3 5	2

Figure 2: Inverted lists for 2-grams of the "username" field.

We can answer a string-similarity query by computing the n-grams of the query string and retrieving the inverted lists of these grams. We then process the inverted lists to find all string ids that occur at least T times, since a string r within edit distance k of another string s must share at least $T = |G(r)| - k \times n$ grams with s [17]. This problem is called the *T-occurrence problem*. Solving the T-occurrence problem yields a set of candidate string ids. The false positives are removed in a final verification step by fetching the strings of the candidate string ids and computing their real similarities to the query. As an example, given a gram length $n = 2$, an edit distance threshold $k = 1$, and a query string q = "`marla`", Figure 3 illustrates how to find the similar usernames from the data in Figure 1. We first compute the 2-grams of q as {"`ma`", "`ar`", "`rl`", "`la`"} and retrieve the inverted lists of these 2-grams. We consider the records that appear at least $T = 4 - 2 \times 1 = 2$ times on these lists as candidates, which have review-ids 2, 3, and 5. Last, we compute the real similarity for these candidates, and

the review-id 5 is the final answer. Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results, which is called a *corner case*. In the above example, if the threshold is 3, then $T = 4 - 2 \times 3 = -2$, causing a corner case.

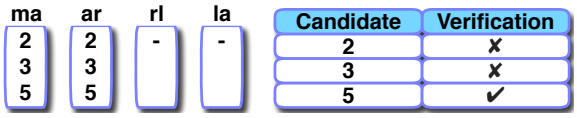


Figure 3: Answering an edit-distance query for "q"=`marla` and T=2.

2.3 Apache AsterixDB

Initiated in 2009, the AsterixDB project integrated ideas from three distinct areas – semi-structured data, parallel databases, and data-intensive computing – to create an open-source software platform that scales on large, shared-nothing commodity computing clusters. AsterixDB consists of several software layers. The topmost layer provides a parallel DBMS with a full, flexible data model (ADM) and query languages (AQL/SQL++) for describing, querying, and analyzing data. The next layer, a query compiler based on *Algebricks* [8], is used for parallel query processing. This algebraic layer receives a translated query plan from the upper layer and transforms it using rule-based optimization. It also generates Hyracks jobs to be executed on the *Hyracks* [6] layer. It provides storage facilities for datasets that are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes [2]. AsterixDB translates a computation into a directed-acyclic graph (DAG) of operators and connectors, and sends it to Hyracks for execution.

Each record in an AsterixDB dataset is identified by a unique primary key and records are hash-partitioned across the nodes on their primary keys. Each partition is locally indexed by a primary key in an LSM B+-tree, a.k.a. the primary index, and resides on its node's local storage. AsterixDB also supports secondary indexing, including B+-tree, R-tree, and inverted indexes, partitioned in the same way as the primary index.

3 USING SIMILARITY QUERIES

In this section, we discuss similarity measures supported in AsterixDB and how users express similarity queries. We also show how users can specify indexes to expedite query processing.

3.1 Supported Similarity Measures

AsterixDB supports two similarity measures, edit distance and Jaccard, to solve string and set similarity queries. Both measures can be processed with or without indexes. Let us focus on edit distance first. It can be calculated on two strings. As an extension in AsterixDB, edit distance can also be computed between two ordered lists. For example, the edit distance between ["`Better`", "`than`", "`I`", "`expected`"] and ["`Better`", "`than`", "`expected`"] is 1. This generalization is possible since a character in a text string can be viewed as an element on an ordered list if we think of a string as a collection of ordered characters.

A Jaccard value can be computed on two lists of elements. If a field type is string, a user can use a tokenization function such as "`word-tokens()`" to make a list of elements from the string. For example, it is possible to calculate the Jaccard similarity between two strings by tokenizing each string into a list of words. The types of the elements on a list should be the same.

If a user wishes to use their own similarity measure, they can opt to create a user-defined function (UDF). A UDF can be expressed in AQL or SQL++ (two query languages supported by AsterixDB) or implemented as an external Java class. If the desired UDF can be expressed in AQL or SQL++, the user can create such a function using the following syntax.

```
create function similarity-cosine(x, y) {
  ......
}
```

3.2 Expressing Similarity Queries

AsterixDB provides two ways to express a similarity query in AQL or SQL++, illustrated by the example AQL in Figure 4. These queries find the record pairs from the Amazon review dataset that have similar summaries. In Figure 4(a) before the actual query, the similarity function and threshold are defined with a "`set`" statement. The query then uses a similarity operator "~=", which is a syntactic sugar defined for similarity functions. The "~=" operator computes the similarity between its two operands according to the "`simfunction`" and "`simthreshold`," and returns the records that are similar. The same query can be written without using the similarity operator by a more experienced user. In Figure 4(b), the query uses the "`similarity-jaccard()`" function, and this query is equivalent to that in Figure 4(a). The first syntax can be easier to use since default settings for "`simfunction`" and "`simthreshold`" exist so that a user does not need to provide the two "`set`" statements. In addition, the user does not need to know the exact function name. Also, if the user wants to change the similarity function, they only need to change the "`set`" statements without changing the query itself. During query parsing and compilation, it is easy for the optimizer to detect this syntactic sugar and generate a desired optimized plan. On the other hand, the second form gives the user more direct control. There are a few variations of similarity functions in AsterixDB, e.g., one that can do early termination during the evaluation, and a user can freely choose any of them.

```
use dataverse TextStore;
set simfunction 'jaccard';
set simthreshold '0.5';
for $t1 in dataset AmazonReview
for $t2 in dataset AmazonReview
where word-tokens($t1.summary) ~= word-tokens($t2.summary)
return { 'summary1': $t1, 'summary2': $t2 }
```

(a) ~= Notation

```
use dataverse TextStore;
for $t1 in dataset AmazonReview
for $t2 in dataset AmazonReview
where similarity-jaccard(word-tokens($t1.summary),
                         word-tokens($t2.summary)) >= 0.5
return { 'summary1': $t1, 'summary2': $t2 }
```

(b) Function Notation

Figure 4: AQL join on the "summary" field of the Amazon reviews using Jaccard similarity.

3.3 Using Indexes

Without an index, AsterixDB scans the whole dataset to compute the result for the given query. To expedite the execution, AsterixDB supports two kinds of inverted indexes. The first type, called "keyword index," uses the elements of a given unordered list, and is suitable for Jaccard similarity. The two queries in Figure 4 could utilize a keyword index on the "summary" field. A keyword index can be created using the following DDL statement, where "smix" is the index name:

```
create index smix on AmazonReview(summary) type keyword;
```

The second index type is "n-gram index," and is suitable for edit distance. An n-gram index uses the extracted n-grams of a string as the keys, and maps those keys to their corresponding primary ids. The following is an example DDL statement to create a 2-gram index on the "reviewerName" field:

```
create index nix on AmazonReview(reviewerName) type ngram(2);
```

4 EXECUTING SIMILARITY QUERIES

In this section, we discuss how similarity queries are internally executed in AsterixDB. First, we present the execution flow for a similarity query in the presence of an index, then describe the execution flow when no index is available.

4.1 Executing Similarity Selections

We first present the execution strategy for selection queries. We use an example query to explain the execution flow for Figure 5, which computes the edit distance between a field V of a dataset and a constant C.

```
for $t1 in dataset bar
where edit-distance($t1.V, C) < 2
return {"id": $t1.id, "field":$t1.V}
```

Figure 5: A similarity-selection query.

4.1.1 Index-Based Search Execution. When running the above query on a cluster with multiple nodes, the query coordinator (*a.k.a.* cluster controller) sends a request containing the constant search key C to each participating node, since AsterixDB uses a shared-nothing architecture. Figure 6 illustrates how a similarity-selection query is executed using a secondary inverted index on a 3-node cluster. Each node contains a partitioned primary index and a local inverted index.

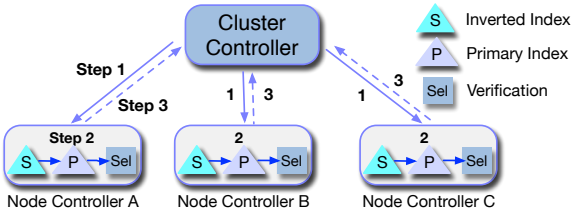


Figure 6: Parallel execution of a similarity-selection query.

If an index is available, AsterixDB runs an index-based selection plan at each node. It first gives the constant value C to the secondary inverted index. The secondary-inverted-index search generates $\langle SecondaryKey, PrimaryKey \rangle$ pairs that satisfy the T-occurrence condition, which may include false positives. It then looks up these primary keys in the primary index to fetch their corresponding records. The primary keys are sorted prior to this search to increase the chance of page cache hits in the buffer. After the primary-index search, a SELECT operator is applied to remove false positives and generate the final results. If the similarity condition is selective enough, such an index-based search plan can be much more efficient than a non-index-based plan that uses SCAN and SELECT operators. Once the local results are generated at each node, they are sent to the coordinator to be combined.

The compiler generates a non-index-based selection plan (the left part of Figure 7). The optimizer then transforms the initial plan to an index-based selection plan if there is an applicable index. We will discuss this rewriting process further in Section 5.1.

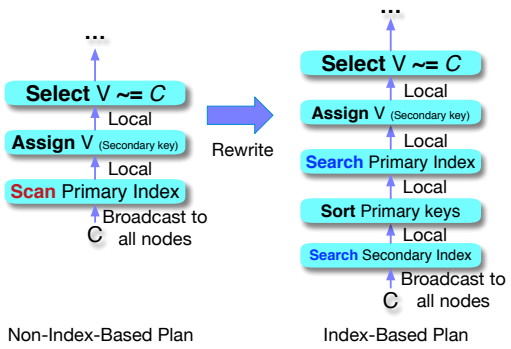


Figure 7: A similarity-selection query plan

4.1.2 Non-Index-Based Search Execution. Similar to index-based execution, when there are multiple nodes, the coordinator sends a request containing the search key C to all the nodes. At each node, as there is no index on the field in the given similarity condition, AsterixDB scans the primary index, fetches all records, and verifies the similarity condition on the given field for each record. This process was depicted on the left part of Figure 7. Finally, the results will be returned to the coordinator.

4.2 Executing Similarity Joins

A join has two branches as its input. We call the first one the "outer branch" and the second one the "inner branch." For example, in Figure 8, the AQL variable $\$t1$ refers to the outer branch and $\$t2$ refers to the inner branch.

```
for $t1 in dataset bar
for $t2 in dataset foo
where similarity-jaccard($t1.A, $t2.B) > 0.5
return {"of1": $t1.f1, "of2": $t1.f2, "A": $t1.A,
        "if1": $t2.f1, "if2": $t2.f2, "B": $t2.B}
```

Figure 8: A similarity-join query.

4.2.1 Index-Based Join Execution. Similar to the similarity-selection case, where a predicate was broadcast to all nodes, the records from the outer branch of each node are broadcast to all nodes in the similarity-join case. Figure 9 depicts how a similarity-join query is executed using a secondary inverted index on a 3-node cluster.

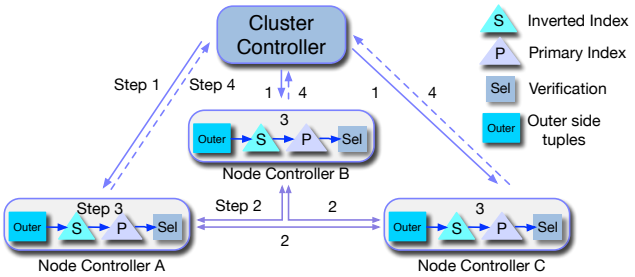


Figure 9: Parallel execution of a similarity-join query.

The coordinator sends the query request to each participating node. Each node of an outer-branch partition scans the outer-branch data and broadcasts its records to all nodes with a secondary-index partition. This broadcast replicates all records of the outer-branch on each node where the secondary-index search will be performed. Each node with an index-side partition uses the incoming outer-branch records as well as its local ones to search

its local inverted index. Once each secondary-index partition has processed all the records from the outer branch, the resulting primary keys from the search will be fed into the primary index, and a primary-index search will be executed. Again, the primary keys are sorted before the primary-index search to increase the chance of page cache hits. As before, we need to remove false positives from the index-based subplan using a SELECT operator on the original similarity condition, which is taken from the join operator. This plan is depicted on the right part in Figure 10. Finally, the results are sent to the coordinator to be combined.

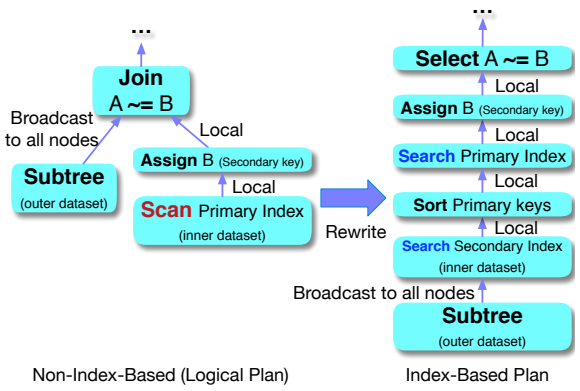


Figure 10: A similarity-join query plan.

4.2.2 Non-Index-Based Join Execution. When there is no index, a simple nested-loop join could be performed. The outer branch would feed the predicate from each record to the primary index of the inner branch. The complexity of this solution is quadratic. To avoid a costly nested-loop join, we instead adopt a three-stage algorithm [34] in AsterixDB.

Since this algorithm uses a prefix-filtering method, a global token order needs to be established to generate a prefix for each field value. This global token order can be any arbitrary order, and we choose the increasing token-frequency order, which tends to generate fewer candidate pairs [34]. The first stage computes a global token order by counting the frequency of each token in the tokenized data and sorting the tokens based on their frequencies. In the second stage, the algorithm computes a short prefix subset for each set based on the global token order produced in the first stage. Then, the record id and only the join attribute of each record are replicated and repartitioned by hashing its prefix tokens. After the repartitioning step, candidate pairs are generated by grouping the pairs by their ids, and the similarity is computed for each pair to filter out the dissimilar ones. This stage produces only similar record id pairs. Finally, the third stage rescans the inputs to fetch the rest of the record fields for these id pairs.

To apply this algorithm in AsterixDB, rather than implementing new operators and complex query plans, we chose to represent the algorithm using existing AQL constructs such as "`for`", "`let`", "`group by`", and "`order by`" since this approach is more extendable in the future. In addition, if/as we improve existing operators, we do not need to modify the given AQL to utilize the improved operators. Figure 11 shows an AQL query capturing the three stages for a self-similarity join on the "summary" field of the Amazon Review dataset, using Jaccard similarity with a threshold; each step is implemented using basic AQL constructs and functions. We now discuss the details of these three stages.

Stage 1: *Token Ordering* is expressed in lines 11-18 of Figure 11. In this subquery we iterate over the records in the dataset. For each record, we retrieve the tokens in the "summary" field and

```
1  // -- - Stage 3 - --
2  for $ARevLeft in dataset('ARevs')
3  for $ARevRight in dataset('ARevs')
4  for $ridpair in
5          // -- - Stage 2 - --
6          for $ARevLeft in dataset('ARevs')
7          let $lenLeft := len($ARevLeft.summary)
8          let $tokensLeft :=
9                  for $tokenUnranked in $ARevLeft.summary
10                 for $tokenRanked at $i in
11                         // -- - Stage 1 - --
12                         for $t in dataset('ARevs')
13                         let $id := $t.ARev_id
14                         for $token in word-tokens($t.summary)
15                         /*+ hash */
16                         group by $tokenGrouped := $token with $id
17                         order by count($id), $tokenGrouped
18                         return $tokenGrouped
19                 where $tokenUnranked = /*+ bcast */ $tokenRanked
20                 order by $i
21                 return $i
22         for $prefixTokenLeft in subset-collection($tokensLeft, 0,
23         prefix-len-jaccard($lenLeft, .5f) - $lenLeft + len($tokensLeft))
24
25         for $ARevRight in dataset('ARevs')
26         let $lenRight := len($ARevRight.summary)
27         let $tokensRight :=
28                 for $tokenUnranked in $ARevRight.summary
29                 for $tokenRanked at $i in
30                         // -- - Stage 1 - --
31                         for $t in dataset('ARevs')
32                         let $id := $t.ARev_id
33                         for $token in word-tokens($t.summary)
34                         /*+ hash */
35                         group by $tokenGrouped := $token with $id
36                         order by count($id), $tokenGrouped
37                         return $tokenGrouped
38                 where $tokenUnranked = /*+ bcast */ $tokenRanked
39                 order by $i
40                 return $i
41         for $prefixTokenRight in subset-collection(
42         $tokensRight, 0, prefix-len-jaccard($lenRight, .5f))
43
44         where $prefixTokenLeft = $prefixTokenRight
45         let $sim := similarity-jaccard($tokensLeft, $tokensRight, .5f)
46         where $sim >= .5f and $ARevLeft.ARev_id < $ARevRight.ARev_id
47         group by $idLeft := $ARevLeft.ARev_id,
48                 $idRight := $ARevRight.ARev_id with $sim
49         return {'idLeft': $idLeft, 'idRight': $idRight, 'sim': $sim[0]}
50
51 where $ridpair.idLeft = $ARevLeft.ARev_id and
52         $ridpair.idRight = $ARevRight.ARev_id
53 order by $ARevLeft.ARev_id, $ARevRight.ARev_id
54 return {'left': $ARevLeft, 'right': $ARevRight, 'sim': $ridpair.sim}
```

Figure 11: Three-stage set-similarity algorithm expressed in AQL for a self join on the Amazon Review (ARevs) dataset using Jaccard similarity with a threshold of 0.5.

count the number of occurrences of each token using a group-by clause. To expedite this calculation, we use a compiler hint in line 15, which suggests using hash-based aggregation instead of the default sort-based aggregation for the group-by statement, since the order of tokens at this particular step is not meaningful. Finally, we order the tokens based on their count using an order-by clause. The same subquery is repeated later, in lines 30-37, in the context of the second dataset. During the optimization, the optimizer will detect the common subquery and execute the subquery only once using a replicate operator to send the results to both outer plans. More details can be found in Section 5.4.2.

Stage 2: *Record ID (RID)-Pair Generation* is expressed in lines 5-50. We scan the dataset in line 6, then retrieve each token from the "summary" field. We order the tokens by the rank computed in the first stage (lines 12-23) by joining the set of tokens in one summary with the set of ranked tokens. We use a hint in line 19 that advises the compiler to use a broadcast join operator to

broadcast the ranked-tokens. Next, we order the join results by rank, stored in the variable "$*i*." We then extract the prefix tokens in line 22, and use the "`prefix-len-jaccard()`" built-in function to compute the length of the prefix for Jaccard similarity with a threshold of 0.5. The built-in "`subset-collection()`" function extracts the prefix subset of the tokens. The same process of tokenizing, ordering the tokens, and extracting the prefix tokens is done in lines 25-42 for the second stream of the dataset. We then join the two streams on their prefix tokens in line 44, and compute and verify the similarity of each joined pair. We use the built-in "`similarity-jaccard()`" function to compute the similarity. Since a pair of records can share more than one token in their prefixes, duplicate pairs could be produced, and they are eliminated by using a group-by clause in line 48.

Stage 3: *Record Join* is expressed in lines 1-4 and 51-54, which consists of two joins. The first join adds the record information for the first RID of each RID pair, while the second join adds the record information for the second.

The plan resulting from this large AQL query is shown in Figure 12. In the figure, "Hash repartition" means that a tuple is repartitioned to a corresponding node based on its hashed value. With "Hash repartition merge," a step of merging tuples based on sort field values occurs after a "Hash repartition." To transform a logical plan generated from a user's similarity join query to the three-stage-similarity query plan utilizing the AQL in Figure 11, we develop a new framework called AQL+, which will be discussed in Section 5.2.

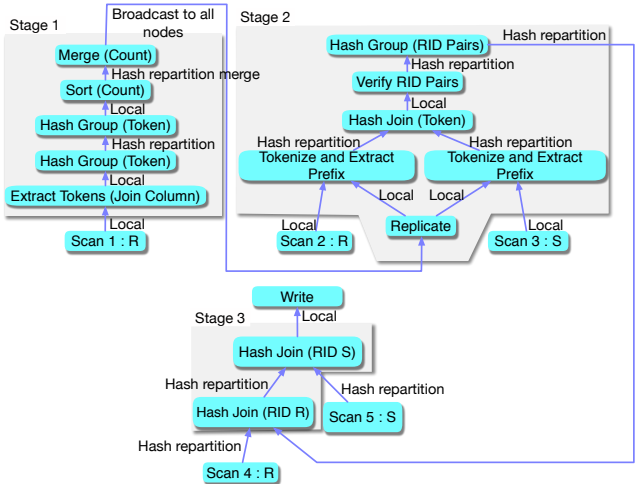


Figure 12: A logical plan of a three-stage-similarity join.

5 OPTIMIZING SIMILARITY QUERIES

In this section, we discuss how AsterixDB optimizes similarity queries and describe the AQL+ framework.

5.1 Rewriting a Similarity Query

AsterixDB uses rule-based optimization [8]. A logical plan is constructed from a given query, and each optimization rule is tried on this plan. If a rule is applicable, then the plan is transformed. A logical plan involving a dataset always starts with a primary-index scan, followed by a SELECT operator if there is one or more conditions. A non-index similarity query plan is constructed first, and an index-based transformation or a three-stage-similarity join can be introduced during the optimization.

5.1.1 Rewriting a Similarity-Selection Query. Figure 7 showed how a similarity-selection query is optimized to use an index. The left-hand side shows the original scan-based query plan, and the right-hand side shows the optimized plan. Based on a selection operator with a similarity condition (using the "~=" notation), the optimizer tries to replace the primary-index scan with a secondary-index-based search plan.

To rewrite a similarity-selection query, the optimizer first matches an operator pattern consisting of a pipeline with a SELECT operator and a PRIMARY-INDEX SCAN operator. Next, it analyzes the condition of the given SELECT operator to see if it contains a similarity condition and if one of its arguments is a constant. If so, it determines whether the non-constant argument originates from the PRIMARY-INDEX SCAN operator and whether the corresponding dataset has a secondary index on a field variable V. For each secondary index on V, it checks an index-to-function-compatibility table (Figure 13) to determine its applicability. For example, an n-gram index can be utilized for the "`edit-distance()`" function. The final SELECT operator filters out false positives.

Index Type	Supported Functions
n-gram	edit-distance(), contains()
keyword	similarity-jaccard()

Figure 13: Index-function compatibility table.

Corner cases: Recall that for queries using edit distance, the lower bound on the number of common q-grams (or tokens) may become zero or negative. For such a corner case, the optimizer must revert to a scan-based plan even if an index is available. For selection queries, it can foresee such cases at compile time when applying the corresponding index-rewrite rule by analyzing the constant argument in the similarity condition. When detecting a corner case, it simply stops rewriting the plan. Note that no such corner cases are possible for similarity queries based on Jaccard, because if two sets have no elements in common, then they can never reach a Jaccard similarity greater than 0. In contrast, two strings could be within a certain (large) edit distance even if the n-gram sets of the (short) strings have no common elements.

5.1.2 Rewriting a Similarity-Join Query. The basic rewriting of a similarity-join query using an index is shown in Figure 10. The optimized plan on the right-hand side uses an index-nested-loop join strategy. Similar to the rewrite for selection queries, the optimizer replaces the primary-index scan of the inner branch with a secondary-index search followed by a primary-index search. Thus, it is required that the inner branch of the join is a primary-index scan, while the outer branch could be an arbitrary operator subtree (shown as "Subtree" in the figure). In the optimized plan, the outer branch feeds into the secondary-index search operator, i.e., every record from "Subtree" will be used as a search key to the secondary index.

As in the similarity-selection case, the optimizer needs to remove false positives from the index-based subplan with a SELECT operator on the original similarity condition, which is taken from the join operator. Notice the "broadcast" connection between the outer subtree and the secondary-index search, which signifies that each partition executing the "Subtree" plan will broadcast its output stream's records to all the secondary-index partitions. The optimizer first matches the required operator pattern consisting of a JOIN that has at least one input coming from a PRIMARY-INDEX SCAN. Next, it analyzes the join condition to make sure the similarity function has two non-constant arguments. If so, it continues

by checking if the inner argument B of the similarity condition is produced by the join input from the PRIMARY-INDEX SCAN, and whether the corresponding dataset has applicable secondary indexes. Finally, the optimizer consults the index compatibility matrix to decide whether it can rewrite the query using an index.

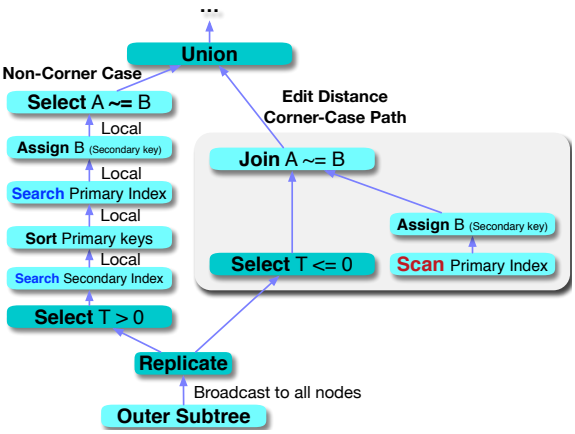


Figure 14: An optimized similarity-join query plan with the corner case.

Corner cases: For string-similarity joins using edit distance, we must modify the basic index-nested-loop join plan in Figure 10 to correctly handle corner cases. Unlike selection queries where the secondary-index search key is a constant, the secondary-index search keys for an index-nested-loop join are produced by the outer branch ("Subtree"). Join corner cases must therefore be dealt with at query runtime, as opposed to query compile time for selection queries. Figure 14 shows the modified index-nested-loop plan for correctly handling corner cases for edit distance. The main difference lies in separating the records produced by the outer subtree into two sets, one containing non-corner-case records ($T > 0$), and one containing corner-case records ($T \leq 0$). We do this by using a replicate operator above the outer subtree, followed by a selection operator on each of its two outputs to filter out the corner-case and non-corner-case records, respectively. The non-corner-case records are fed into the secondary-to-primary index plan as before, while the corner records participate in a non-index nested-loop join plan. The final query answer is the union of the results of those two joins.

5.2 AQL+ Framework

As discussed in Section 4.2.2, we need to find a way to transform a nested-loop-join plan generated from a user's similarity-join query to a three-stage join plan. An issue is that, unlike the index-nested-loop-join optimization that adds or replaces a few operators from a nested-loop join plan, as we can see in the AQL query in Figure 11, a three-stage-similarity join query generates a large number of operators. Figure 15 shows the number of operators in a three-stage-similarity join.

Operator	Count
Assign	12
Data-Scan	2
Join	1
Total	**15**

Nested-loop join plan

Operator	Count	Operator	Count
Aggregate	6	Join	3
Assign	44	Order	3
Data-Scan	6	Select	4
Group	3	Unnest	8
		Total	77

Three-stage-similarity join plan

Figure 15: Number of operators for a nested-loop join and three-stage-similarity join plan for the same query.

Due to this complexity, it would be difficult to build an optimization rule that manually constructs these operators to transform a simple nested-loop join plan to a three-stage join plan. Instead, we develop a novel rewrite framework called "AQL+." As shown in Figure 16, we use this framework to convert a simple logical plan generated from a user's join query to a three-stage join plan.

Once the optimizer receives a logical plan in AQL+, it extracts the information from the logical plan and integrates it into an AQL+ query template. The generated AQL+ query can be parsed and compiled again using the AQL+ parser and translator. The result is a transformed logical plan, and the plan optimization process can then continue.

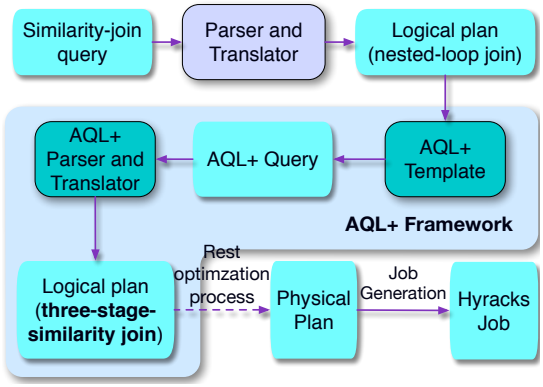


Figure 16: Execution of a similarity-join query using AQL+.

To combine the information from a logical plan and the three-stage-similarity-join AQL query template, we need to find ways to refer to the portions of the logical plan from the query template. Therefore, the AQL+ framework consists of a few AQL language extensions and the compilation of these language extensions during the optimization process. As a result, the AQL+ language is a superset of AQL. The AQL+ has three AQL extensions: Meta Variable (denoted as "$$"), Meta Clause ("##"), and Explicit Join ("join"). We need these extensions to refer to the logical variables and operators in the logical plan during the optimization process, since the AQL+ transformation of a given plan happens during the optimization process. This is because the optimizer only sees the logical plan and physical plan, not the original query. Since AQL itself does not have an explicit join clause, AQL+ includes one in order to express a join on two branches. For example, we use meta-variables to refer to the primary keys of the input records or variables in the similarity predicate. The usage of meta-clauses is to refer to inputs of the AQL query and to logical constructs that cannot be directly specified in AQL, such as joins. So any AQL+ template can be combined with any join input branches, where the inputs can be from any kind of subplans of other algebraic operators. In addition, to support various types of data, similarity functions, and thresholds, the similarity-join rule template uses placeholders, which are parts of the AQL+ query and are unknown until runtime. They are used for data types, similarity-specific functions, or values. For example, the "SIMILARITY" placeholder is used for built-in AQL functions, and the "THRESHOLD" placeholder is for numerical values.

Table 1: AQL+ extensions.

Extension	Symbol	Functionality
Meta Variable	$$	Refer to a variable in the plan
Meta Clause	##	Refer to an operator in the plan
Join Clause	join	Express an explicit join

The AsterixDB optimizer integrates the information from the given logical plan into the AQL+ query template and compiles the

resulting AQL+ query. Specifically, for a three-stage-similarity join, it needs to identify a similarity join operator that contains a Jaccard similarity join and its threshold. It also needs to get the information about the two branches of this join operator. Using the information from the join operator, the logical plan fed into this AQL+ template can be transformed to the equivalent three-stage-similarity plan. Rather than doing this transformation by introducing a number of operators by hand, we rely on the existing compilation path to generate a revised plan. This process is depicted in Figure 16; the details of this optimization will be discussed in the next subsection.

For the similarity join query in Figure 11, the optimizer will generate an equivalent AQL+ template and use it to transform a simple query during the rule-rewrite phase. In this way, the simple query of Figure 4(a) can be transformed to the query in Figure 11 during the optimization process. Figure 17 shows a part of the AQL+ template that generates a three-stage-similarity-join plan. Here, we can see that actual dataset-scans are replaced with meta-clauses and a meta-variable ($$*LEFTPK*_3) is used to refer to to the primary key of an incoming record in the given logical plan. Join-clauses are used to join two meta-clauses.

```
1  //---Stage3---
2  join( (##RIGHT_1),
3        ( join( (##LEFT_1) ,
4  // -- - Stage 2 - --
5                ( join( (##LEFT_2
6  ......
7                            //--- Stage1---
8                            ##LEFT_3
9                            let $id := $$LEFTPK_3
10                           for $token in TOKENIZER($$RIGHT_3)
11                           /*+ hash */
12                           group by $tokenGrouped := $token with $id
13 ......
```

Figure 17: A part of three-stage-similarity-join algorithm expressed in AQL+.

In addition, the AQL+ framework can be applied to transform multi-way-similarity join plans as well because of its power to handle a logical plan iteratively. Similar to non-similarity-join cases, multi-way-similarity joins can be transformed sequentially without a limitation. For instance, Figure 18 shows a similarity-join plan involving four datasets. The join between first two datasets, R and S, has already been transformed into a three-stage-similarity join plan. This branch will act as the outer branch when the optimizer processes the next join operator on the third dataset T.

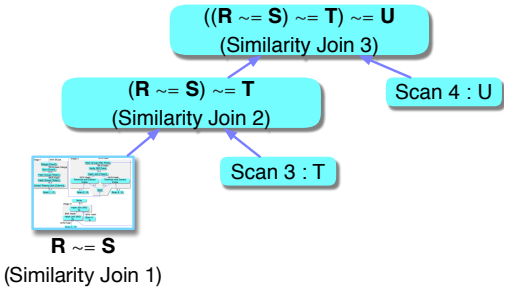


Figure 18: Rewriting a multi-way-similarity-join plan on four datasets.

It should be noted that AQL+ is a general extension framework, not only for similarity queries, and it can be used to support a transformation using AQL during the compilation process.

5.3 Optimization Rule For Similarity Queries

As discussed before, optimization in AsterixDB is rule-based [8]. Once the Algebricks layer receives a compiled plan from an AQL query, it first optimizes the given plan logically. Then Algebricks sets up physical operators for each logical operator. After that, the physical optimization phase begins. When it is finished, a Hyracks job is created and executed. During the logical and physical optimization, there are a number of rule sets that are applied sequentially. A rule can be assigned to multiple rule sets. Based on the configuration of a rule set, each rule can be applied repeatedly until no rule in the set can transform the plan.

To apply the similarity-query optimization framework to this optimization path, we create a new rule set for the AQL+ framework and similarity queries. The rule set includes a *similarity join rule* (SJR) along with a handful of other rules that need to be applied after SJR is applied. As described earlier, the main functionality of AQL+ is a transformation using a complex AQL template to re-generate a logical plan while maintaining the current surrounding plan as part of the new plan. SJR first analyzes the conditions of each join operator. If its condition includes a similarity predicate, it applies the AQL+ template to the plan to generate an AQL+ query. Then it compiles the query into a new logical Algebricks plan. Some parts of the plan were already optimized if they belonged to the original incoming plan. However, most part of the plan is not optimized yet, since the three-phase plan was just compiled and has not gone through the optimization process before the SJR rule set. Therefore, the newly generated plan needs to go through some of the earlier optimization rules again. This re-application process is not necessary for non-similarity queries, since the plan generated from non-similarity queries is not transformed in the SJR rule set. Therefore, we need to ensure that the similarity-join rule set is only applied to similarity-join queries. The benefit of this approach is that the optimization for similarity queries can be processed without interfering with non-similarity queries. This approach also gives a chance to the newly generated similarity-query plan to reach the same level of transformation once the similarity rule set has finished its work.

5.4 Improvements

We discuss two improvements to similarity query processing, which can be applied to non-similarity query processing as well.

5.4.1 Surrogate Index-Nested-Loop-Join. A drawback of an index-nested-loop join using a local secondary index is the need to broadcast the outer side data to all secondary-index partitions as explained in Section 4. For example, during an execution of the AQL query in Figure 8, the outer side needs to broadcast join key field "A," as well as "$f1$" and "$f2$" field. If there are more fields in the return clause, the broadcasting cost will be increased as well. This broadcast step is a direct consequence of the co-partitioning of each secondary index with its primary index. Also a secondary-inverted-index search can generate multiple pairs of results for the same primary key, as there can be multiple entries of the secondary keys for the same primary key; thus, we also want to reduce the sorting cost between the secondary-index search and primary index search. We can reduce the cost by only sending the secondary-key fields together with a compact surrogate for each outer-side record, so that we can later use the surrogates to obtain the surviving original records. This idea is reminiscent of semi-join optimization in distributed databases [40].

Figure 19 shows a surrogate-based index-nested-loop-similarity join plan. Notice the PROJECT operator that follows the REPLICATE operator after the outer subtree, which eliminates all non-essential fields from the outer side. The optimizer filters out the "$f1$" and "$f2$" fields since the search key is "A." In addition, since

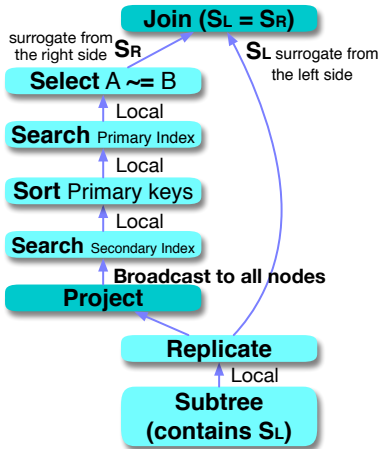


Figure 19: Surrogate index-nested-loop-join plan.

the same subtree is used twice in the plan, a REPLICATE operator is introduced to reduce the subtree calculation time. We will discuss this optimization in-depth in the next subsection. After the secondary-to-primary index search, we must use the surrogates from the outer side to obtain their complete records. As shown in the figure, we resolve the surrogates via a top-level join of the original outer subtree with the indexed nested-loop subtree (after removing false positive matches). Since the top-level join is an equi-join on the surrogates S_L and S_R, it can be executed efficiently in parallel, e.g., using a hash join.

5.4.2 Materializing/Reusing Shared Subplans. As shown in a simplified version of a three-stage-similarity join in Figure 20, in case of the three-stage-similarity self join, the dataset R may need to be scanned three to four times. For this case, we could simply execute the original data-scan operation four times. However, if the two branches of this join result from a complex computation from a subquery, it would be expensive to compute the result of the subquery many times. To minimize the cost, AsterixDB materializes the common subplan and reuses it several times.

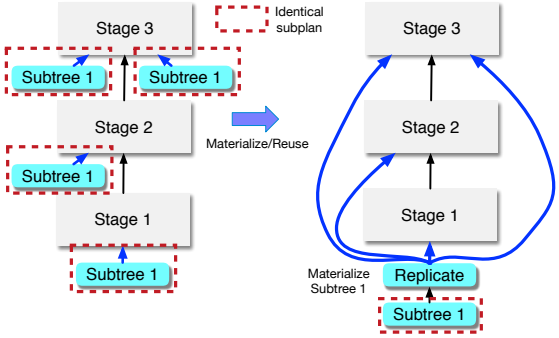


Figure 20: Materializing and reusing a subtree of a three-stage-similarity self join.

6 EXPERIMENTS

We have conducted an experimental evaluation of our approach in AsterixDB using large, real data sets. We used an 8-node cluster to host an AsterixDB (0.9.2) instance, where each node ran Ubuntu with a Quadcore AMD Opteron CPU 2212 HE (2.0GHz), 8GB RAM, 1 GB Ethernet NIC, and two 7,200 RPM SATA hard drives. Each dataset was horizontally partitioned into 16 partitions (2 per node) based on primary keys to provide full I/O parallelism. Table 2 shows the AsterixDB configuration parameters.

Table 2: AsterixDB parameters for the experiments.

Parameter	Value
Global memory budget per node	6GB
Budget for in-memory components per dataset	1.5GB
Data page size	128KB
Disk buffer cache size	2GB
Sort buffer size	128MB
Join buffer size	128MB
Group-by buffer size	128MB

6.1 Datasets

Different similarity functions were used for different types of data. Edit distance is more suitable for short string fields, while Jaccard is more suitable for long fields with many elements. To evaluate AsterixDB for different similarity functions, we used three datasets with different characteristics, as shown in Table 3. The Amazon Review dataset, discussed in earlier sections, included Amazon product reviews [23]. The Reddit Submission dataset included Reddit postings for about eight years. The Twitter dataset had 1% of US tweets for three months obtained via Twitter's public API. When imported into AsterixDB, each data set had an additional auto-generated primary key field, as AsterixDB requires that each dataset must have a primary key. Other than this field, we did not define more fields in the schema. This gave us a lot of flexibility to import any datasets into AsterixDB. The dataset size in AsterixDB was greater than the raw data size, since each record included the information about each field. For example, for a string field, its type needs to be included in addition to its value.

Table 3: Dataset properties.

Dataset	AmazonReview	Reddit	Twitter
Content	Amazon product reviews	Reddit postings	Tweets
Number of Records	83.68M	196M	155M
Data Period	1996 - 2014	01/2006 - 08/2015	06/2016 - 08/2016
Raw Data Format	JSON	JSON	JSON
Raw Data Size	55 GB	252 GB	465 GB
Dataset Size	60.6 GB	320 GB	582 GB
Fields used	summary, reviewerName	title, author	text, user.name

Table 4 shows the characteristics of the fields of the datasets. The minimum character length and minimum word count of the fields were 0. The first three fields were used for edit distance, while the last three fields were used for Jaccard.

Table 4: The characteristics of the fields.

Field	Avg char count	Max char count	Avg word count	Max word count
AmazonReview. reviewerName	10.3	49	1.7	14
Reddit.author	24.3	275	4.1	32
Twitter.user.name	10.6	20	1.7	10
AmazonReview. summary	22.8	361	4.0	44
Reddit.title	1,056.2	400K	1,173	20K
Twitter.text	62.5	140	9.7	70

6.2 Index Size

We built a keyword index for Jaccard similarity queries and a 2-gram index for edit distance queries. To measure the execution time of basic exact match queries on the same fields as a baseline, we also built a B+ tree index on the search fields. Table 5 shows

the index sizes for the Amazon Review dataset and the time it took to create each index. An *n*-gram index took much more space than a B+ tree or keyword index as it had more secondary keys per record. For instance, a 2-gram index on the "reviewerName" field took 15.6GB of disk space, which was about 25% of the original dataset size. The size of a keyword index was also greater than a B+ tree index on the same field since there are many secondary keys per record. For a given type of index, the construction time was roughly proportional to the size of an index. In each case, the dataset itself was also stored in a primary B+ tree index.

Table 5: Index size and build time for Amazon reviews.

Field	Index Type	Size (GB)	Build Time (s)
Dataset itself	B+ tree	60.6	1,563
reviewerName	B+ tree	2.7	223
reviewerName	2-gram	15.6	1,441
summary	B+ tree	3.5	275
summary	keyword	5.4	573

6.3 Selection Queries

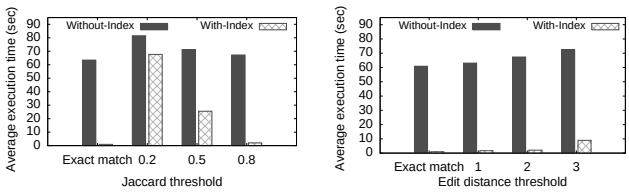
To measure the performance of similarity-selection queries, we first created a search value set that contained 10,000 random unique values extracted from the search field. For Jaccard queries, we ensured that the minimum number of words in each value in the set was 3. For edit distance queries, the minimum length of characters in each value was 3. For each similarity threshold, we randomly chose a search value from the set for each query, sent 100 such queries to the cluster, and measured the average execution time. The performance baseline for comparison purposes was an equality-condition query that used the same value for the given field. The query template in Figure 21 below was used to measure the average execution time. "Simfunction" and "simthreshold" in the queries were replaced with a specific similarity function and a threshold. "V" was the given field and "C" was the random value from the above set.

```
count ( for $o in dataset X
        where @simfunction($o.V, C) >= @simthreshold
        return {"oid":$o.id, "v":$o.V} );
```

Figure 21: Similarity-selection query template.

6.3.1 Jaccard Similarity. For each of the three datasets we ran similarity queries using Jaccard similarity on suitable fields using different thresholds: 0.2, 0.5, and 0.8. Figure 22(a) shows the results. We see that the average execution time for similarity selection queries decreased as the threshold increased in case of index-based plans. For example, it took the index-based method 67.6 seconds to conduct a Jaccard query with a threshold of 0.2, while it only took 25.5 seconds to execute a query with a threshold of 0.5. If there was no applicable index, both similarity and exact-match queries showed a high execution time as each record had to be read from the primary index and that scan time was a dominant factor in the overall execution time. We can also see the overhead of the similarity query versus the exact-match query for all the thresholds since it takes more time to calculate a Jaccard value than to get the result of an exact match. This overhead decreased as the threshold increased; this is because we applied certain optimizations such as early termination and pruning based on string lengths, which significantly reduced the cost of computing the similarity.

When the threshold was low, the times were similar for both index-based and non-index-based queries. This is because the candidate set size using T-occurrence for index-based queries was quite large when the threshold was low. This can be seen in



(a) Jaccard (b) Edit distance

Figure 22: Execution time of selection queries on Amazon reviews.

Table 6. As the number of candidates increased, the search time increased due to the need for a primary-index lookup for each candidate.

Table 6: Candidate size and the final result size for the indexed-select query for Amazon reviews in Figure 22(a).

Jaccard Threshold	Actual Result Record Count (B)	Candidate Set Record Count (C)	Ratio (B/C)
0.2	559,167	8,298,473	6.7%
0.5	12,260	660,016	1.9%
0.8	36	12,420	0.3%

6.3.2 Edit Distance. We measured the average execution time of an edit distance selection query using different thresholds, namely 1, 2, and 3. Figure 22(b) shows the results. As the threshold increased, the execution time increased. The reason is similar to the case of Jaccard queries: the candidate set size using T-occurrence increased as the threshold increased. It took the index-based method 2 seconds to run a selection query with a threshold of 2; it took 8.9 seconds to run a query with a threshold of 3. We can also see that the execution time of non-index-based edit distance queries increased as the threshold increased for the same reason as described above.

6.4 Join Queries

To measure the performance of similarity join queries, we ran self-join queries on the three datasets. Specifically, the query template in Figure 23 was used to measure the average execution time as in the similarity-selection query case. Here, V is the field on which we applied a similarity function and *id* is the primary key field.

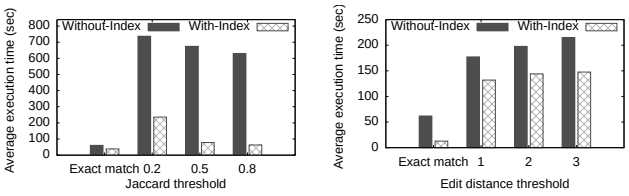
```
count ( for $o in dataset X
        for $i in dataset X
        where @simfunction($o.V, $i.V) >= @simthreshold
        and $o.f1 = C and $o.id < $i.id
        return {"oid":$o.id} );
```

Figure 23: Similarity-join query template.

6.4.1 Varying Threshold. We first extracted certain number of records from the outer branch of the join to limit its input. For each query, we chose 10 random records from the outer branch. In the query template in Figure 23, a field named $f1$ was used to specify such a limit. For Jaccard join queries, we used different thresholds, namely 0.2, 0.5, and 0.8. For edit distance queries, we used thresholds of 1, 2, and 3. When there was no applicable index, AsterixDB chose to employ the three-stage-similarity-join plan for Jaccard queries. The results are shown in Figures 24(a) and 24(b). The trends were similar to those of selection queries except for the exact-match join, which significantly outperformed both Jaccard

and edit distance joins since it used a hash join, where the join keys were broadcast to multiple nodes.

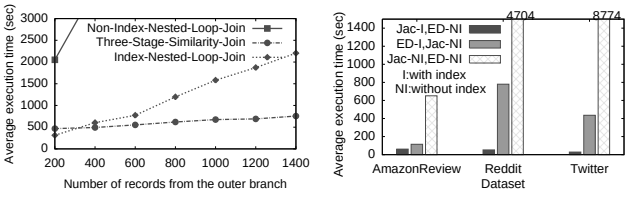
Regarding the compilation overhead of AQL+, we observed that the average overhead of generating a new logical three-stage-similarity-join plan using AQL+ for the queries in Figure 24(a) was around 50 ms, and it took around 500 ms to optimize that plan. The overall compilation time of the three-stage-similarity-join query was around 900 ms.



(a) Jaccard (b) Edit distance

Figure 24: Execution time of join queries on Amazon reviews.

6.4.2 Varying Record Number. For a Jaccard join query, its execution time was smallest when the threshold was 0.8. In this experiment, we varied the number of records from the output branch and fixed the threshold at 0.8. The times for the non-index-nested-loop self-join, index-nested-loop self-join, and three-stage-similarity self-join on the Amazon Review dataset are shown in Figure 25(a). We increased the number of output records from the outer branch and measured the resulting execution time of each join. First, we see that the execution time of non-index-nested-loop self-join was already highest for 200 records and increased drastically compared to other two types of joins. Once the number of output records from the outer branch reached around 400, the three-stage-similarity join began to outperform the index-nested-loop join. This is because the time for the index-nested-loop join is proportional to the number of records fed to the secondary-index search, as it needs to deal with each record at a time. In contrast, for the three-stage-similarity join, most of the time is spent on global-token-order generation in the first stage. Once this is generated and broadcast to all the nodes, hash joins in stage 2 and 3 can deal with the incoming records efficiently, since each join key is sent to only one node. This benefit is visible in the figure. For instance, the time for the three-stage-similarity join for 800 records was 619 seconds, while it was 674 seconds for 1,000 records. This result shows only 55 seconds of increase, while the execution-time difference for the index-nested-loop joins going from 800 to 1,000 records was 384 seconds.



(a) Similarity joins on Amazon reviews (b) Multi-way-join queries on the three datasets

Figure 25: Execution time of join queries.

6.4.3 Multi-Way Join Queries. So far, we have used only one similarity condition per similarity query. Next, we used two similarity conditions in a query and varied the order of the two conditions. The query template in Figure 26 was used to measure the average execution time. The dataset Y and the field $f1$ were used to limit the number of records from the outer branch.

```
count ( for $p in dataset Y
        for $o in dataset X
        for $i in dataset X
        where $p.f1 = $o.f1 and $p.id = C
        and @simfunction1($o.V1, $i.V1) >= @simthreshold1
        and @simfunction2($o.V2, $i.V2) <= @simthreshold2
        and $o.id < $i.id
        return {"oid":$o.id, "iid":$i.id} );
```

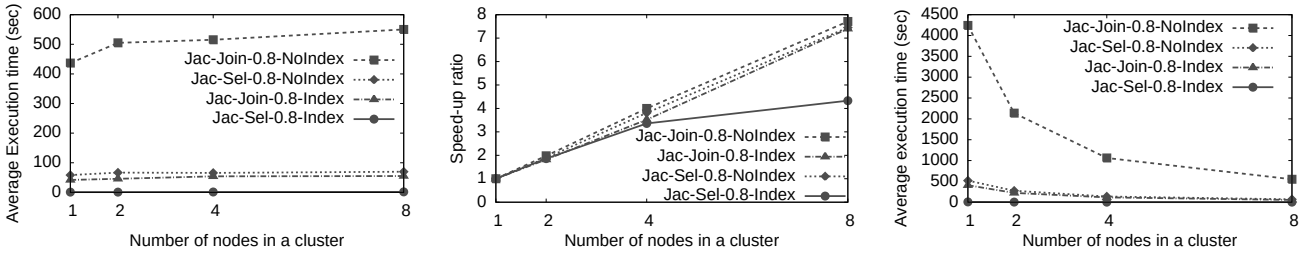
Figure 26: Multi-way-join query template.

Each query had an equi-join and a similarity join with two conditions, including a Jaccard condition with a threshold of 0.8 and an edit distance condition with a threshold of 1. For the equi-join, we used an index-nested-loop join to fetch output records quickly. Also, this join was used to limit the number of output records from the outer branch fed into the similarity join. Then, Jaccard similarity and edit distance conditions were applied. If we applied the Jaccard condition first, the Jaccard join will be followed by the edit distance condition in a SELECT operator. For both similarity conditions, we used an index-based method for the first condition and a non-index-based method for the second. Figure 25(b) shows that the performance was the best when the index-based-Jaccard join was conducted first, as there were no corner cases for Jaccard similarity. This order generated fewer candidates than applying the index-based edit distance predicate first. In contrast, for the edit distance case, it needed to augment the corner-case path in the logical plan, thus generated more candidates. In addition, it should be noted that other queries showed similar patterns for all the three datasets as well. That is, the average execution time of a similarity query was proportional to the size of datasets when the result cardinality was similar.

6.5 Scalability Tests

6.5.1 Scale-Out. For the scale-out experiment, we used four clusters with different sizes, namely 1, 2, 4, and 8 nodes. When we doubled the number of nodes in a cluster, we also doubled the data size to store the same amount of data per node. Thus, the 1-node cluster had 12.5% of our original data set size, the 2-node cluster had 25%, and the 4-node cluster had 50% of the data. The 8-node cluster contained the original dataset, where the data size was 100%. In other words, each node had 12.5% of the original dataset. Ideally, the response-time graph would show a flat line per query. As the number of nodes increased, the queries were handled as expected, as shown in Figure 27(a). We can see some variance in the case of the Jaccard-similarity join without an index; in the three-stage-similarity join, the global token order generated in stage 1 of the join needed to be broadcast to all the nodes. Therefore, as the number of nodes increased, the communication cost increased as well. This gap was the greatest between 1 node and 2 nodes, since that was where we first incurred the communication cost of global-token-order propagation. However, the execution time increase was not high between 2 nodes and 4 nodes. Between 4 nodes and 8 nodes, we can see the trend as well. Once the communication cost was accounted for, the execution time of three-stage-similarity joins was quite scalable.

6.5.2 Speed-Up. For the speed-up experiment, we also used four cluster sizes (1, 2, 4, and 8 nodes) with each cluster size



(a) Scale-out for Jaccard (b) Speed-up on Jaccard (c) Times for Jaccard speed-up

Figure 27: Scale-out and speed-up queries on Amazon reviews.

being given the entire (100%) data set. Figure 27(b) shows that the speed-up was proportional to the number of nodes. The speed-up for the index-based-Jaccard-selection query with a threshold 0.8 was less than that of the other Jaccard queries. This was because its execution time was already less than a few seconds on the 1-node cluster, and there was a basic overhead for each cluster such as communication cost. In particular, the execution time of that query on the 1-node cluster was 6.5 seconds, and its execution time on the 8-node cluster was 1.5 seconds. Figure 27(c) shows the execution time of the same queries on each cluster.

7 CONCLUSIONS

In this paper, we presented the support for similarity queries in Apache AsterixDB, a parallel data management system. We described the entire life cycle of a similarity query in the system, including the query language, indexing, execution plans with or without index, and plan rewriting to optimize the execution. Our solution leverages the existing infrastructure of AsterixDB, including its operators, query engine, and rule-based optimizer. We presented an experimental study based on several large, real data sets on a parallel computing cluster to evaluate the proposed techniques, and showed their efficacy and performance to support similarity queries on large data sets using parallel computing.

Acknowledgments The Apache AsterixDB project has been supported by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, 0844574, and by NSF CNS awards 1305430 and 1059436. This work was also sponsored by the National Science Foundation of China under grants 61572373, 60903035, and 61472290, and by the National High Technology Research and Development Program of China under grant 2017YFC08038. The project has received industrial support from Amazon, eBay, Facebook, Google, HTC, Infosys, Microsoft, Oracle Labs, and Yahoo! Research.

REFERENCES

[1] Peter Christen. 2012. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. SSBM.
[2] Alsubaiee et al. 2014. Storage Management in AsterixDB. In *Proceedings of the 2014 VLDB Conference*.
[3] Bayardo et al. 2007. Scaling up all pairs similarity search. In *Proceedings of the 2007 WWW Conference*.
[4] Borgatti et. al. 2009. Network analysis in the social sciences. *Science* (2009).
[5] Behm et al. 2009. Space-Constrained Gram-Based Indexing for Efficient Approximate String Search. In *Proceedings of the 2009 ICDE Conference*.
[6] Borkar et al. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 ICDE Conference*.
[7] Bouros et al. 2012. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment* (2012).
[8] Borkar et al. 2015. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proc. of the ACM Symp. on Cloud Computing*.
[9] Ciaccia et al. 1997. M-tree: An E cient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 1997 VLDB Conference*.
[10] Chaudhuri et al. 2006. Data Debugger: An Operator-Centric Approach for Data Quality Solutions. *IEEE Data Eng. Bull.* (2006).
[11] Deng et al. 2014. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Proceedings of the 2014 ICDE Conference*.
[12] Deng et al. 2014. A pivotal prefix based filtering algorithm for string similarity search. In *Proceedings of the 2014 SIGMOD Conference*.
[13] Doulkeridis et al. 2014. A survey of large-scale analytical query processing in MapReduce. *Proceedings of the VLDB Endowment* (2014).
[14] Feng et al. 2012. Trie-join: a trie-based method for efficient string similarity joins. *Proceedings of the VLDB Endowment* (2012).
[15] Gravano et al. 2001. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of the 2001 VLDB Conference*. 491–500.
[16] Gravano et al. 2003. Text joins in an RDBMS for web data integration. In *Proceedings of the 2003 WWW Conference*.
[17] Jokinen et al. 1991. Two algorithms for approxmate string matching in static texts. In *International Symposium on Mathematical Foundations of Computer Science*.
[18] Jiang et al. 2014. String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment* (2014).
[19] Li et al. 2007. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In *Proceedings of the 2012 VLDB Conference*. http://www.vldb.org/conf/2007/papers/research/p303-li.pdf
[20] Li et al. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *Proceedings of the 2008 ICDE Conference*.
[21] Metwally et al. 2012. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment* (2012).
[22] Mann et al. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins.. In *Grundlagen von Datenbanken*.
[23] McAuley et al. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 2015 SIGKDD Conference*.
[24] Mann et al. 2016. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment* (2016).
[25] Minghe et al. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* (2016).
[26] Qin et al. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceeding of the 2011 SIGMOD Conference*.
[27] Rahm et al. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* (2000).
[28] Ribeiro et al. 2011. Generalizing prefix filtering to improve set similarity joins. *Information Systems* (2011).
[29] Sarawagi et al. 2004. Efficient set joins on similarity predicates. In *Proceedings of the 2004 SIGMOD Conference*.
[30] Silva et al. 2010. The similarity join database operator. In *Proceedings of the 2010 ICDE Conference*.
[31] Silva et al. 2012. Exploiting mapreduce-based similarity joins. In *Proceedings of the 2012 SIGMOD Conference*.
[32] Silva et al. 2015. Similarity Joins: Their implementation and interactions with other database operators. *Information Systems* (2015).
[33] Sun et al. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *Proceedings of the VLDB Endowment* (2017).
[34] Vernica et al. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proceedings of the 2010 SIGMOD Conference*.
[35] Wang et al. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 SIGMOD Conference*.
[36] Wang et al. 2013. Scalable all-pairs similarity search in metric spaces. In *Proceedings of the 2013 ACM SIGKDD Conference*.
[37] Wang et al. 2013. VChunkJoin: An efficient algorithm for edit similarity joins. *KDE, IEEE Transactions on* (2013).
[38] Xiao et al. 2008. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. In *Proceedings of the 2008 VLDB Conference*.
[39] Xiao et al. 2008. Efficient similarity joins for near duplicate detection. In *Proceedings of the 2008 WWW Conference*.
[40] Donald Kossmann. 2000. The state of the art in distributed query processing. *ACM Comput. Surv.* (2000).