Information Systems 88 (2020) 101455

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Similarity query support in big data management systems

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

[a] University of California, Irvine, CA, USA
[b] Wuhan University, Wuchang, Wuhan, Hubei, China

ARTICLE INFO

Article history:
Received 25 May 2019
Received in revised form 9 October 2019
Accepted 9 October 2019
Available online 14 October 2019
Recommended by Dennis Shasha

Keywords:
Similarity query
Parallel database
Optimization

ABSTRACT

Similarity query processing is becoming increasingly important in many applications such as data cleaning, record linkage, Web search, and document analytics. In this paper we study how to provide end-to-end similarity query support natively in a parallel database system. We discuss how to express a similarity predicate in its query language, how to build indexes, how to answer similarity queries (selections and joins) efficiently in the runtime engine, possibly using indexes, and how to optimize similarity queries. One particular challenge is how to incorporate existing similarity join algorithms, which often require a series of steps to achieve a high efficiency, including collecting token frequencies, finding matching record id pairs, and reassembling result records based on id pairs. We present a novel approach that uses existing runtime operators to implement such complex join algorithms without reinventing the wheel; doing so positions the system to automatically benefit from future improvements to those operators. The approach includes a technique to transform a similarity join plan into an efficient operator-based physical plan during query optimization by using a template expressed largely in the system's user-level query language; this technique greatly simplifies the specification of such a transformation rule. We use Apache AsterixDB, a parallel Big Data management system, to illustrate and validate our techniques. We conduct an experimental study using several large, real datasets on a parallel computing cluster to assess the similarity query support. We also include experiments involving three other parallel systems and report the efficacy and performance results.

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 [2], and social media analysis [3]. 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 with a title 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. A plethora of data structures, partitioning schemes, and algorithms have been developed to support similarity queries efficiently on large datasets. 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.

Since in many applications data resides in a database, a natural question is how to adopt these existing techniques on such a database system to support similarity queries. One approach is to use these techniques *on top of* the database. That is, we develop an independent application layer that retrieves data from the

* Correspondence to: Department of Computer Science, University of California, Irvine, CA, 92697, USA.
E-mail addresses: taewok2@uci.edu (T. Kim), lwh@whu.edu.cn (W. Li), alexander.behm@gmail.com (A. Behm), icetindil@gmail.com (I. Cetindil), rvernica@gmail.com (R. Vernica), vinayakb@gmail.com (V. Borkar), mjcarey@ics.uci.edu (M.J. Carey), chenli@ics.uci.edu (C. Li).

[1] Part of his work was done when visiting UC Irvine.
[2] Work done when affiliated with UC Irvine.

https://doi.org/10.1016/j.is.2019.101455
0306-4379/

database, and deploy these indexing structures and algorithms in this applications to support similarity queries. One advantage of this approach is that it has a lot of flexibility in the implementation. Meanwhile, it also has several major drawbacks. First, the data essentially has two copies, one inside the database, and one in the application. Second, additional effort is needed to synchronize the data in the application with the data in the database, in order to return the latest results to a user query. Third, the internal capabilities of the database, including storage, indexing, and query execution, are not fully utilized. Another approach is to fully integrate these techniques *inside* a database, so that all the abovementioned limitations can be overcome. In particular, the data does not have to be copied in a separate layer, and queries can be supposed on the data directly by utilizing the built-in capabilities of the database system.

In this paper, we focus on the second approach, and study how to support similarity queries *end-to-end* in a full, declarative parallel data management system natively. By "end-to-end", we mean the whole lifecycle of a query, including the language support for similarity conditions, internal index structures, execution plans with or without an index, plan rewriting to optimize execution, and so on. Achieving the end-to-end goal has several challenges. First, as the notion of similarity in queries can be domain specific, we need to support commonly used similarity functions and let 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 database operators without "reinventing the wheel" (i.e., without introducing new, ad hoc operators). Third, we need to consider how to leverage an existing query optimization framework to rewrite similarity queries to achieve high performance.

In this paper, to develop and validate our approach, we use Apache AsterixDB, an open-source parallel data management system for semi-structured (NoSQL) data, as an example platform. We make the following contributions:

(1) We show how to extend the existing query language of a parallel database system 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 the existing operators in this parallel database system, both for index-based and non-index-based plans (Section 4) and for both search and join queries. Our solution not only allows query plans to benefit from the built-in optimizations in those operators but also automatically enjoys future improvements in the operators.

(3) We show how to rewrite similarity queries in the existing rule-based optimization framework inside AsterixDB (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 [4] can involve up to 77 operators (Section 5.2). To allow the optimizer to more easily transform such complex plans, we develop 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 the parallel database system to support queries with more than one similarity join condition, making AsterixDB the first parallel data management system (to our best knowledge) to support similarity queries with multiple similarity joins.

(4) We present an empirical study using several large, real datasets on a parallel cluster to evaluate the effects of these techniques. We also present results of comparative experiments with two other parallel systems to explore the relative efficacy of AsterixDB's support for parallel similarity queries on large datasets when our technique has been incorporated (Section 6).

An earlier version of this paper appeared in [5]. This article extends the earlier version by adding: (1) a thorough description of the AQL+ optimization framework and the inclusion and an explanation of the actual template that transforms a simple nested-loop similarity join into a three-stage similarity join; (2) a more detailed description of several optimization techniques; (3) a substantially extended experimental section including: (3a) a re-execution of all the experiments from [5] using AsterixDB's new SQL++ language, (3b) the evaluation of multi-way similarity joins on real datasets, (3c) the results of using two additional real large datasets, (3d) doubling the scale of the speed-up and scale-out experiments, (3e) an analysis of new findings from the new scale experiments, (3f) empirical performance comparisons with three other big data management systems; and (4) additional diagrams and extended discussions throughout the paper.

1.1. Related work

There are various kinds of similarity queries on strings and sets. Many algorithms (e.g., [6–8]) use a gram-based approach for string similarity search. VGRAM [9] extends the approach by introducing variable-length grams. To optimize string similarity joins, filtering techniques are widely used. Length filtering uses the length of a string to reduce the number of candidates. An example algorithm is gram-count [10]. Prefix filtering [11–20] utilizes the fact that two strings can be similar only if they share some commonality in their prefixes. Based on this fact, many algorithms have been proposed, such as AllPair [11], PPJoin/PPJoin+ [12], ED-Join [16], MPJoin [14], QChunk [18], VChunk [19], and Pivotal prefix [20]. Other algorithms have been proposed such as M-Tree [21], trie-Join [22], and partition-based set-similarity join [23].

There have been several evaluation studies of string/similarity [24] and set-similarity joins [25]. There was also a recent survey paper about string similarity queries [26]. The authors of [24] found that AdaptJoin [17] and PPJoin+ [12] were best for Jaccard similarity. Meanwhile, the authors of [25] concluded that AllPair [11] was still competitive. The authors of [26] discussed prefix-filtering techniques. Many of these algorithms assume that the data to be searched or joined fits into main memory.

For parallel similarity joins, a number of studies have used the MapReduce framework [4,27–31]. There was also a survey that discussed parallel MapReduce similarity joins [32]. Vernica et al. [4] proposed a three-stage algorithm in such a setting. There have also been studies on integrating similarity joins into database management systems [10,33–36]. Some of these adopted the similarity join as a UDF [10] or expressed a similarity join in a SQL expression [33]; others have introduced a relational operator to support similarity joins [34–36].

Our focus in this paper is different, as it is about supporting similarity queries in a general-purpose parallel database system setting. We need to address various systems-oriented challenges when adopting existing techniques in this context. A parallel similarity query processing system called Dima [37] was also proposed recently. Dima is an in-memory-based system, however, and our focus is on handling big data that cannot fit into memory. There are some other search systems and DBMSs that support similarity queries, including Elasticsearch, Oracle, and Couchbase. 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 that work for both select and join operations, and similarity predicates can be part of a general declarative query along with non-similarity predicates.

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

Fig. 1. Example data of Amazon Review dataset (simplified).

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

Fig. 2. Inverted lists for 2-grams of the `username` field.

ma	ar	rl	la
2 3 5	2 3 5	-	-

Candidate	Verification
2	✘
3	✘
5	✔

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

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 available 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 on how 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`"}.

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. Fig. 2 shows the inverted lists for the 2-grams of the `username` field of the little sample Amazon Review dataset in Fig. 1.

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 [38]. This problem is called the *T-occurrence problem*. Solving the T-occurrence problem yields a set of candidate string ids. The false positives are then 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`", Fig. 3 illustrates how to find the similar usernames from the data in Fig. 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; this 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.

2.3. Apache AsterixDB

Initiated in 2009, the AsterixDB [39] 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 [40].

2.3.1. AsterixDB architecture

AsterixDB consists of several software layers as shown in Fig. 4. The top-most layer provides a full, flexible data model (ADM) and query languages (SQL++ [41] and AQL) for describing, querying, and analyzing data. AQL was AsterixDB's initial query language; SQL++ is now the preferred language for users.

The next layer, a query compiler based on *Algebricks* [42], is used for parallel query processing. This algebraic layer receives a translated logical SQL++/AQL query plan from the upper layer and performs rule-based optimizations. 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 further transform the plan. For logical plan transformation, there are currently 15 rule sets and 171 rules (including multiple assignments of a rule to different rule sets). After logical optimization, Algebricks selects physical operators for each logical operator in the plan. For example, for a logical join operator, a hybrid-hash-join or nested-loop-join can be chosen based on the join predicate. After that, the physical optimization phase begins. During logical and physical optimization, there are a number of rule sets that are applied sequentially. There are 3 rule sets and 30 rules for the physical optimization phase. Once the physical optimization process is done, the Algebricks layer generates Hyracks jobs to be executed on the *Hyracks* [43] layer.

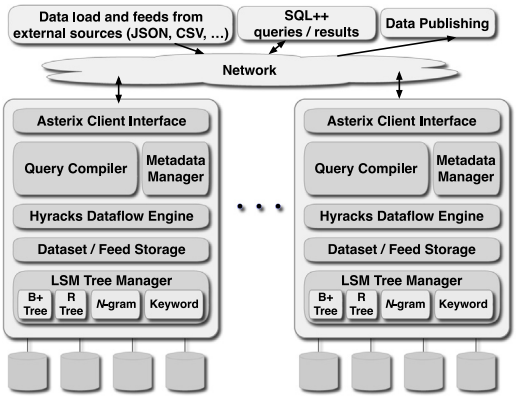


Fig. 4. AsterixDB architecture.

```
create dataverse exp;
use exp;

create type AmazonReviewType as open {
  review_id: int,
  username: string,
  review_summary: string
}

create dataset AmazonReview(AmazonReviewType)
primary key review_id;
```

Fig. 5. An ADM type and dataset.

The Hyracks layer includes the storage facilities for datasets that are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes [44]. AsterixDB translates a computation task into a directed-acyclic graph (DAG) of operators and connectors and sends it to Hyracks for execution. In Hyracks, operators consume partitions of input data and produce partitions of output data. The output partitions produced by operators are then repartitioned by connectors to produce the input partitions for the next operator. An operator has one or more activities (sub-steps or phases) and there may be control dependencies between two activities on certain operators. Using this information, one or more stages are created. A stage includes a group of activities (an activity cluster) that can be co-scheduled. Therefore, a job will be executed on a stage-by-stage-basis. Since data is represented as tuples of bytes at this level, Hyracks is a data-model agnostic layer.

2.3.2. AsterixDB data model

AsterixDB defines its own data model (ADM) [39] targeting semi-structured data. ADM is a superset of JSON, with support for bags, nested types, and a variety of primitive types. Fig. 5 shows some example ADM DDL including the type definition for the Amazon review dataset in Fig. 1. AmazonReviewType is defined as an open type, which means that its instances must have all the specified fields but may also contain extra fields that can vary from instance to instance.

Each record in an AsterixDB dataset is identified by a unique primary key, and records are hash-partitioned across the nodes of a cluster based on their primary keys. Each record in a dataset has to comply with its associated datatype. Fig. 5 also includes a SQL++ statement for creating the Amazon review dataset. 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 index options; indexes are local, i.e., they are partitioned in the same way as the primary index. Like the primary index, each secondary index also adopts an LSM-based structure. Further details of LSM-based index structures in AsterixDB can be found in the AsterixDB storage management paper [44].

3. Using similarity queries

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

3.1. Supported similarity measures

AsterixDB currently supports two built-in similarity measures, Jaccard and edit distance, to solve set-similarity and string queries. Both measures can be processed with or without indexes. Let us focus on edit distance first. This measure can be calculated on two strings. As an extension in AsterixDB, edit distance can also be computed between two arrays of scalar values. 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 in an array if we think of the string as a collection of ordered characters.

The other supported measure, the Jaccard value, can be computed on two arrays or multisets of elements. If a field type is string, a user can use a tokenization function to first make an array of elements from the string. For example, it is possible to calculate the Jaccard similarity between two strings by tokenizing each string into an array of words.

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

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

3.2. Expressing similarity queries

AsterixDB provides two ways to express a similarity query in a SQL++ or AQL query, both illustrated by the example SQL++ queries in Fig. 6. These equivalent queries find the record pairs from the Amazon review dataset that have similar summaries. In Fig. 6(a) before the actual query, the similarity function and threshold are defined with `set` statements. The query then uses a similarity operator ~=, which is syntactic sugar defined for similarity functions. This similarity 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 also be written without using the similarity operator. The similarity query in Fig. 6(b) uses a Jaccard function named `similarity_jaccard()`, and this query is equivalent to that in Fig. 6(a). The first syntax can be easier to use because the `simfunction` and `simthreshold` also have the default settings and a user is not required to provide the two `set` statements. In addition, the user does not need to remember the exact function

```
use TextStore;

// method 1 - similarity parameters
set simfunction 'jaccard';
set simthreshold '0.5';
select element {"summary1":t1, "summary2": t2}
from AmazonReview t1, AmazonReview t2
where word_tokens(t1.summary) ~= word_tokens(t2.summary);

// method 2 - functions
select element {"summary1":t1, "summary2": t2}
from AmazonReview t1, AmazonReview t2
where similarity_jaccard(word_tokens(t1.summary),
                         word_tokens(t2.summary)) >= 0.5;
```

Fig. 6. SQL++ join on the summary field of the Amazon review dataset using Jaccard similarity.

name with that syntax. 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 replace 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. A user can freely choose any of them.

3.3. Using indexes

Without an index, AsterixDB scans the whole dataset in the query to compute the result for the given query. To expedite query execution, AsterixDB supports two kinds of inverted indexes to support the two similarity measures efficiently.

The first index type, called keyword index, uses the elements of a given multiset as keys and maps those keys to their corresponding primary ids. For example, it is possible to tokenize a string and use each token as a key. This index is suitable for Jaccard similarity. The two queries in Fig. 6 could utilize a keyword index on the summary field. A keyword index can be created using the following DDL statement, where summaryidx is the index name:

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

The second index type is called the *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. For example, we can use the following DDL statement to create a 2-gram index on the reviewerName field:

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

4. Executing similarity queries

In this section, we explain how similarity queries are internally executed in AsterixDB. First, we describe the internal structure of the inverted index. After that, we describe the execution flow for a similarity query in the presence of an index and then describe the execution flow in the case where no index is available.

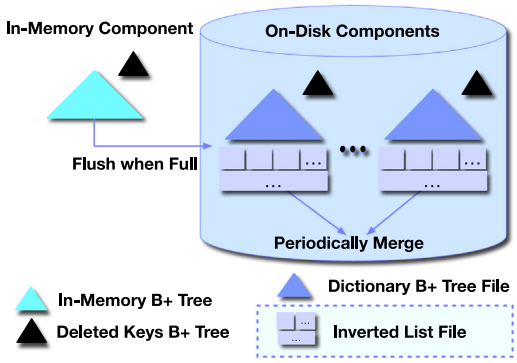
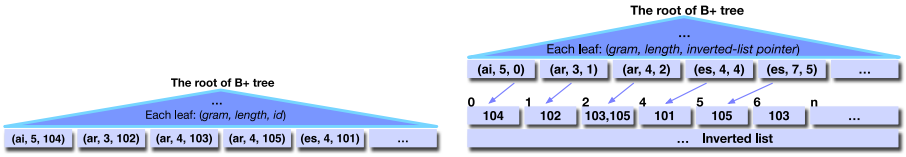


Fig. 7. The structure of an inverted index.

4.1. Inverted index

An inverted index in AsterixDB is an LSM-based secondary index that consists of a mutable in-memory component and multiple immutable on-disk components, as illustrated in Fig. 7. This design choice was made to support high-frequency insertions, as LSM-based indexes amortize the cost of writes by consolidating updates in memory before writing them to disk [44]. The in-memory component consists of two B^+-trees, one for the in-memory inverted index and one to store the primary keys of deleted records. On-disk components are immutable, so AsterixDB denotes the deleted records of the on-disk components using this B^+-tree instead of deleting them from the inverted index itself. This design choice also implies that primary keys obtained from the inverted index may have already been deleted, so they need to be verified by checking their existence in the deleted-key B^+-tree. An in-memory index component grows with inserted/deleted records until the memory budget allocated for the component is exhausted. It is then flushed to disk as a new immutable on-disk component. The multiple index components on disk must be searched besides the in-memory component during a given index-search operation. To mitigate this, AsterixDB periodically merges on-disk components based on a configurable merge policy.

To improve its search performance, AsterixDB employs a length-based technique to partition the inverted index. This technique is useful since it allows the use of length filtering prior to a search, which eliminates records that are not similar based on the length required for the similarity threshold of a query. We use the number of tokens in the given field as the length. Fig. 8(a) shows the details of an in-memory inverted index. The secondary key field is first tokenized based on the type of the index (*n*-gram or keyword), and each token is inserted into the in-memory inverted index along with the length of the secondary key field and the primary key of the record. The in-memory index component is a B^+-tree with keys consisting of triples. Each triple contains $\langle token, length, primary\ key \rangle$. For example, in Fig. 8(a), the leftmost entry is $\langle ai, 5, 104 \rangle$. Its secondary key token is ai and the number of tokens for the given field is 5. The triple also tells us that this entry comes from the record whose primary key is 104. Once the in-memory index is flushed to disk, it becomes immutable, and it is finalized by merging the primary keys with the same token and length into a sorted list in the inverted list file and using the resulting $\langle token, length, inverted\ list\ pointer \rangle$ triples as B^+-tree keys as shown in Fig. 8(b). The pointer there indicates the starting offset of the associated list of primary keys for the given *token* and *length* pair.



(a) In-memory inverted index. (b) On-disk inverted index.

Fig. 8. An example instance of an *n*-gram inverted LSM index.

```
select r.id, r.title
from Reddit as r
where edit_distance(r.title, "good competitions") < 2;
```

Fig. 9. A similarity-selection query.

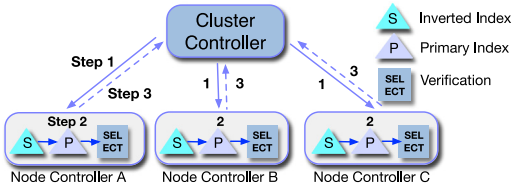


Fig. 10. Parallel execution of a similarity-selection query.

4.2. Executing similarity selections

We first present the execution strategy that AsterixDB uses for selection queries. We use the example query in Fig. 9 to explain the execution flow; this SQL++ query computes the edit distance between a field `title` of a dataset called Reddit and a constant search key `good competitions` where the edit-distance-threshold is 2.

4.2.1. 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 (*a.k.a.* node controller). Fig. 10 illustrates how such a similarity-selection query is executed using a secondary inverted index on a 3-node cluster. In the figure, C is `good competitions` and V refers to the `title` field in Fig. 9. Each cluster node contains a partitioned primary index and a local inverted index. That is, the contents of each inverted index are generated from the local primary index. Thus, the nodes do not need to communicate with each other to execute a 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 ⟨*Secondary Key*, *Primary Key*⟩ 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 lookup to increase the chance of page cache hits in the buffer. After fetching the actual field value from the primary index, 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 the non-index-based plan that uses DATASET-SCAN and SELECT operators in the absence of an index. Once the local results are generated at each node, they are sent to the coordinator to be combined into the final query result.

To process a similarity-selection query, the SQL++ compiler first generates a simple non-index-based selection plan (the left part of Fig. 11) from a user query. The optimizer then transforms the initial plan into an index-based selection plan if there is an applicable index during the logical optimization process. We will discuss this rewriting process further in Section 5.1.

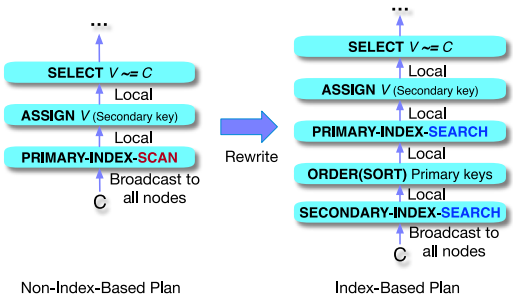


Fig. 11. Similarity-selection query plans.

```
select ar.reviewer_id, ar.id, ar.summary,
       r.author_id , r.id, r.title
from AmazonReview ar, Reddit r
where similarity_jaccard(ar.summary, r.title) > 0.5;
```

Fig. 12. A similarity-join query.

4.2.2. 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. The left part of Fig. 11 depicts this process. Finally, the results will be returned to the coordinator.

4.3. Executing similarity joins

A similarity join operator has two branches as its input. We call the first one the `outer branch` and the second one the `inner branch`. For example, in Fig. 12, the SQL++ alias *ar* refers to the outer branch and *r* refers to the inner branch. This query fetches three fields from each dataset based on a Jaccard join condition with a threshold of 0.5.

4.3.1. Index-based join execution

Similar to the similarity-selection case, where the search predicate value was broadcast to all nodes, in the similarity-join case, the records coming from the outer join branch of each node are broadcast to all nodes. Fig. 13 depicts how a similarity-join query is executed using a secondary inverted index on a 3-node cluster.

The coordinator first sends the query execution request to each participating node. Each node of an outer-branch partition scans its portion of the outer-branch data. While doing so, it broadcasts the resulting records to all nodes with a partition

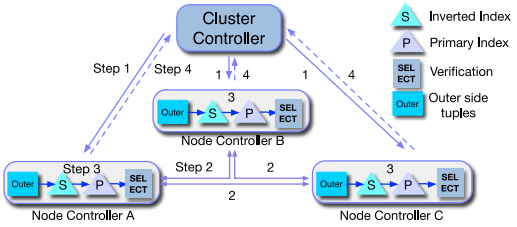


Fig. 13. Parallel execution of a similarity-join query.

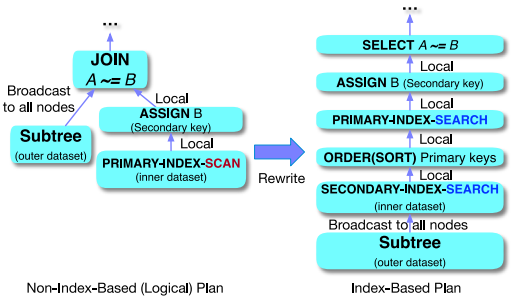


Fig. 14. A similarity-join query plan.

of the inner branch's dataset. This replicates all records of the outer-branch on each node, which then perform a secondary-index search. 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 inner dataset's primary index and a primary-index search will be conducted. As discussed before, these primary search 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 based on the original similarity condition, which is taken from the JOIN operator. The right side of Fig. 14 depicts this process. Finally, the results are sent to the coordinator to be combined.

4.3.2. Non-index-based join execution

When there is no index, a simple nested-loop join could be performed for a similarity join query. The outer branch would feed the predicate from each record to the inner branch. The complexity of this solution would be quadratic. To avoid such a costly nested-loop join, we instead adopt a three-stage-similarity-join algorithm [4] that we review here for ad hoc similarity join processing in AsterixDB.

The three-stage algorithm uses a prefix-filtering method, so a global token order needs to be computed to generate a prefix for each field value. This global token order can be arbitrary; we choose the increasing token-frequency order, which tends to generate fewer candidate pairs [4]. The first stage computes this 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. The record id and (only) the join attribute of each record are then replicated and repartitioned by hashing on these 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 of

```
1  // -- - Stage 3 - --
2  for $ARevLeft in dataset AmazonReview
3  for $ARevRight in dataset AmazonReview
4  for $ridpair in
5   // -- - Stage 2 - --
6   for $ARevLeft in dataset AmazonReview
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 AmazonReview
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 AmazonReview
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 AmazonReview
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}
```

Fig. 15. Three-stage set-similarity algorithm expressed in AQL for a self join on the Amazon Review dataset using Jaccard similarity with a threshold of 0.5.

the algorithm rescans the inputs to fetch the rest of the query's desired record fields for these id pairs.

To apply this three-stage algorithm in AsterixDB, rather than implementing new query operators and complex query plans, we chose to describe the algorithm by using existing AQL constructs such as `for`, `let`, `group by`, and `order by` since this approach would be potentially more extendable in the future. In addition, if/as we improve AsterixDB's existing operators, we would not need to modify the AQL description to utilize the improved operators. For example, if a new sort algorithm becomes available for the sort that generates a global token order, its benefit will be applied without any alteration of the AQL. Fig. 15 shows an AQL query that captures the three stages for a self-similarity join on the `summary` field of the Amazon Review dataset using Jaccard similarity with a threshold. Note how 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 Fig. 15. In this subquery, we iterate over the records in the dataset. For each record, we retrieve the tokens in the `summary` field and 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 that 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 optimization, the optimizer will detect this common subquery and execute the subquery only once by 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 and 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 dataset. We then join the two streams on their prefix tokens in line 44, and compute and verify the similarity of each joined pair using the built-in `similarity-jaccard()` function. Since a pair of records can share more than one token in their prefixes, duplicate pairs can be produced, and they are eliminated by using a group-by clause in line 47.

Stage 3: *Record Join* is expressed in lines 1–4 and 51–54, which consist 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 logical query plan resulting from this large AQL query is shown in Fig. 16. `Hash repartition` in the figure means that a tuple will be repartitioned to a corresponding node based on its hashed value. `Sort merge` repartitioning on a node merges incoming tuples based on their sort field values. To transform a logical query plan generated from a user's SQL++ or AQL query into a similarity join query to the three-stage-similarity query plan similar to the explicit AQL in Fig. 15, we developed a new framework called AQL+, which will be discussed in Section 5.2.

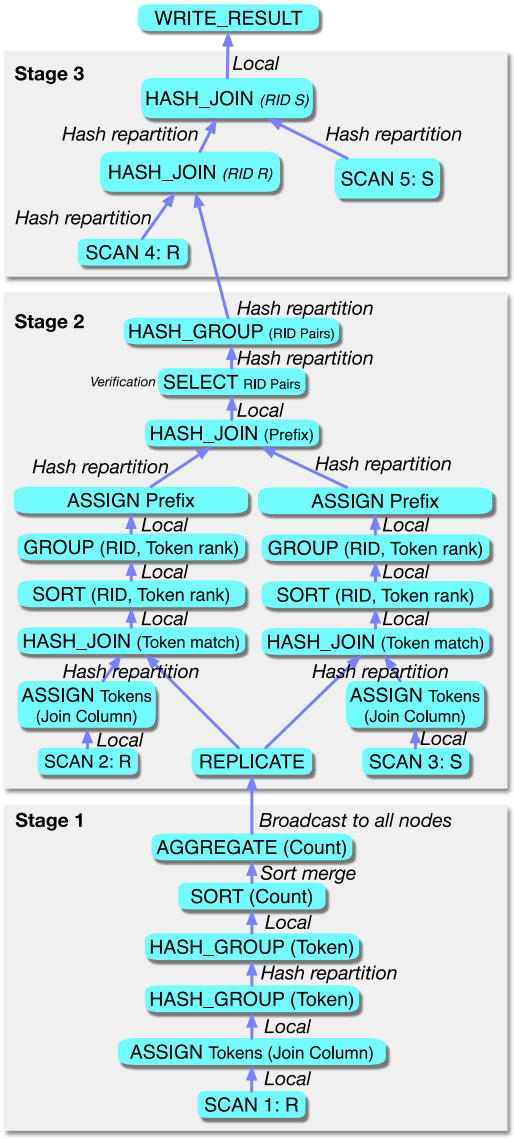


Fig. 16. A plan of a three-stage-similarity join query.

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

Fig. 17. Index-function compatibility table.

5. Optimizing similarity queries

In this section, we discuss how the AsterixDB query processor optimizes SQL++ (or AQL) similarity queries and we describe the AQL+ framework in more detail.

5.1. Rewriting a similarity query

AsterixDB uses rule-based optimization approach [42] as described in Section 2.3.1. An initial logical plan is constructed from a given query, and each optimization rule is tried on this plan. If a rule is applicable, the plan is transformed. A logical plan involving a dataset always starts with a PRIMARY-INDEX-SCAN operator, followed by a SELECT operator if there are one or more conditions. For similarity queries, 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

Fig. 11 from Section 4 shows how a similarity-selection query is optimized to use an index. The left-hand side shows the original scan-based plan, and the right-hand side shows the optimized plan. Based on a SELECT operator with a similarity condition, 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

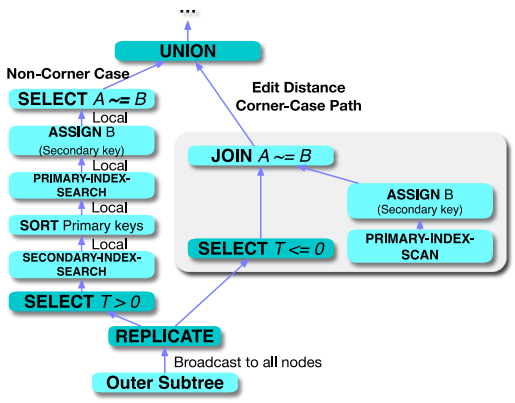
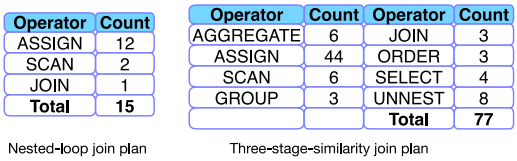


Fig. 18. An optimized similarity-join query plan with the corner case.



Operator	Count
ASSIGN	12
SCAN	2
JOIN	1
Total	**15**

Nested-loop join plan

Operator	Count	Operator	Count
AGGREGATE	6	JOIN	3
ASSIGN	44	ORDER	3
SCAN	6	SELECT	4
GROUP	3	UNNEST	8
		Total	**77**

Three-stage-similarity join plan

Fig. 19. Number of operators for a nested-loop join and three-stage-similarity join plan for the same query.

the corresponding dataset has a secondary index on a field variable V. For each secondary index on V, the optimizer checks an index-function-compatibility table (Fig. 17) to determine its applicability. For example, an n-gram index can be utilized for the `edit_distance()` function. The final SELECT operator in the figure filters out false positives.

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 since an index cannot be used for non-positive T-occurrence values. For selection queries, the optimizer 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 Fig. 14 from Section 4. The optimized query 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 simply 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 using a SELECT operator based 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. This connection tells the `Subtree` to broadcast its output stream's records to all of the inner dataset's secondary-index partitions.

The optimizer first matches the join operator's required pattern, which is a PRIMARY-INDEX-SCAN in the inner branch, since this operator fully scans the dataset rather than using a secondary index on some other condition. Also, the optimizer checks the inner branch since it considers using a secondary index only from the inner branch, not from the outer branch. Next, it analyzes the join condition to make sure the similarity function has two non-constant arguments and checks if an argument of the similarity condition is produced by the PRIMARY-INDEX-SCAN operator in the inner branch, and whether the corresponding inner dataset has an applicable secondary index to support the required similarity lookups. The optimizer then consults the index compatibility matrix to decide whether it can rewrite the query using an index.

Corner cases: For string-similarity joins using edit distance, we must modify the basic index-nested-loop join plan in Fig. 14 to 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 the query runtime, as opposed to the query compile time as for selection queries. Fig. 18 shows the modified index-nested-loop plan for 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 SELECT operators on each of its two outputs to filter out the corner-case and non-corner-case records, respectively. As before, the non-corner-case records are fed into the secondary-to-primary index plan. The corner-case 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 4.3.2, for non-index-based similarity joins, the optimizer needs to transform a nested-loop-join plan generated from a user's query into a three-stage join plan to accelerate similarity-join-query execution. A challenge is that, unlike the index-nested-loop-join optimization that adds or replaces a few operators from a nested-loop join plan, a three-stage-similarity join plan contains a large number of operators as illustrated by the AQL query in Fig. 15. Fig. 19 shows the number of operators in a three-stage-similarity join.

Due to the complexity of the tree-stage-join query plan, it would be rather difficult to build and maintain an optimization rule that manually constructs the DAG of operators that transform a simple nested-loop join plan into a three-stage join plan. Instead, we developed a novel rewrite framework called `AQL+` that converts a simple logical plan generated from a user's similarity-join query into a three-stage join plan. The flow of the AQL+ framework is depicted in Fig. 20. The essential part of the AQL+ framework is the use of an AQL+ query template to express sophisticated query expressions, integrate the information from the incoming logical plan into it, and finally transform the plan during the optimization process. This way the optimizer does not need to have a complex rule that manually translates a simple nested-loop-join plan into a three-stage-similarity-join plan. What we

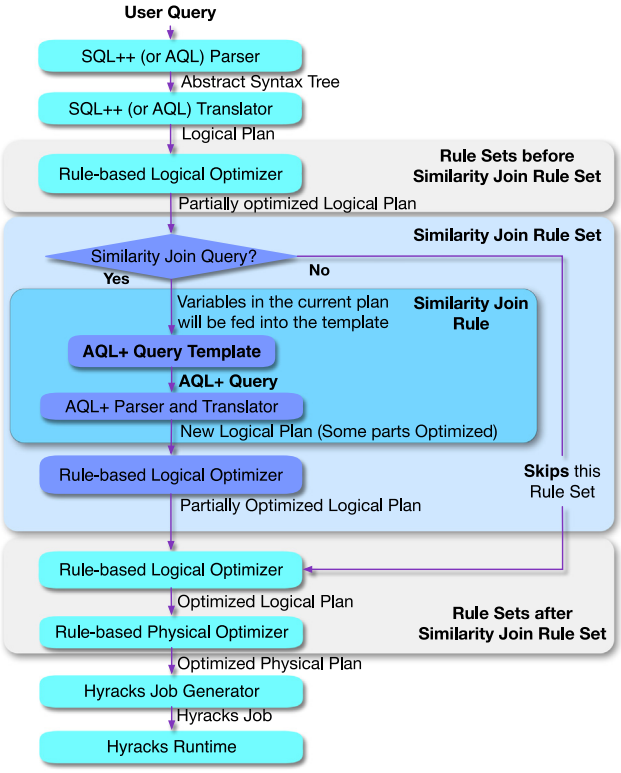


Fig. 20. Execution of a similarity-join query using AQL+.

Table 1
AQL+ extensions (to AQL).

Extension	Symbol	Functionality
Meta Variable	$$	Refer to a variable in the plan
Meta Clause	##	Refer to an operator in the plan
Join Clause	join, loj	Express an explicit inner join or left-outer join

need instead is an AQL+ query template that expresses the three-stage-similarity join and for the AQL+ framework to combine the incoming plan with the query template.

When the SQL++ (or AQL) optimizer receives a logical similarity-join plan in AQL+, it extracts the information from the plan and integrates it with an AQL+ query template that expresses the three-stage-similarity join. The generated AQL+ query is then parsed and compiled again using the AQL+ parser and translator since the generated query itself is also a query. The result of this process is a transformed logical plan. The resulting plan is then processed by the rest of the query plan optimization process.

To combine the information from an incoming logical plan and the three-stage-similarity-join AQL query template, we need ways to refer to relevant portions of the surrounding logical plan from within the AQL+ 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 first AsterixDB query language. (Note that we developed AQL+ when AQL was the primary language of AsterixDB, so AQL+ was based on AQL. We later adopted SQL++ in AsterixDB and SQL++ is now mainly used. However, the differences between SQL++ and AQL do not affect the AQL+ framework since the AQL+ framework works on the logical plan level.) The AQL+ language has three AQL extensions as shown in Table 1: Meta Variable (denoted as $$), Meta Clause (##), and Explicit Join (`join`). We

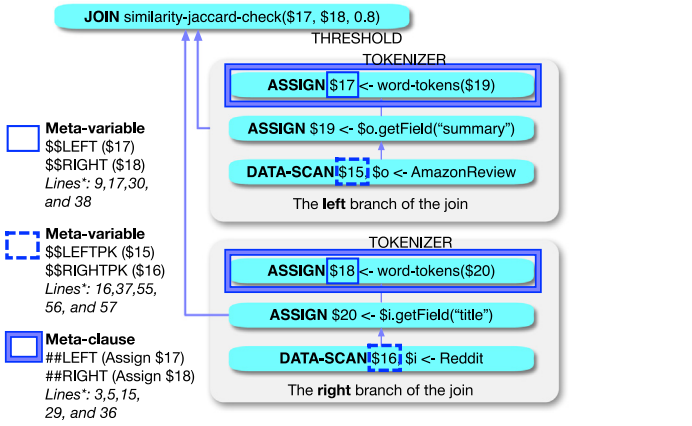
```
1  // --- Stage3 ---
2  join(
3   (##LEFT_0),
4   (join(
5     (##RIGHT_0),
6     // --- Stage 2 ---
7     (join(
8   (##RIGHT_1
9   let $tokensUnrankedRight := TOKENIZER($$RIGHT_1)
10  let $lenRight := len($tokensUnrankedRight)
11  let $tokensRight :=
12   for $token in $tokensUnrankedRight
13   for $tokenRanked at $i in
14    // --- Stage1 ---
15    ##RIGHT_3
16    let $id := $$RIGHTPK_3
17    for $token in TOKENIZER($$RIGHT_3)
18    /*+ hash */
19    group by $tokenGrouped := $token with $id
20    order by count($id), $tokenGrouped
21    return $tokenGrouped
22   where $token = /*+ bcast */ $tokenRanked
23   order by $i
24   return $i
25  for $prefixTokenRight in subset-collection($tokensRight, 0,
26      PREFIX_LEN(len($tokensRight), THRESHOLD))
27    ),
28
29    (##LEFT_1
30  let $tokensUnrankedLeft := TOKENIZER($$LEFT_1)
31  let $lenLeft := len($tokensUnrankedLeft)
32  let $tokensLeft :=
33   for $token in $tokensUnrankedLeft
34   for $tokenRanked at $i in
35    // --- Stage1---
36    ##RIGHT_2
37    let $id := $$RIGHTPK_2
38    for $token in TOKENIZER($$RIGHT_2)
39    /*+ hash */
40    group by $tokenGrouped := $token with $id
41    order by count($id), $tokenGrouped
42    return $tokenGrouped
43   where $token = /*+ bcast */ $tokenRanked
44   order by $i
45   return $i
46  let $actualPreLen:=PREFIX_LEN(len($tokensUnrankedLeft), THRESHOLD)
47                                  - $lenLeft + len($tokenLeft)
48  for $prefixTokenLeft in subset-collection(
49   $tokensLeft, 0, $actualPreLen)
50  ),
51  $prefixTokenLeft = $prefixTokenRight)
52    let $sim := SIMILARITY($tokensRight, $tokensLeft)
53    where $sim >= THRESHOLD
54    /*+ hash */
55    group by $idLeft:=$$LEFTPK_1, $idRight:=$$RIGHTPK_1 with $sim),
56   $$LEFTPK_0 = $idLeft)),
57  $$RIGHTPK_0 = $idRgiht)
```

Fig. 21. Three-stage-similarity join algorithm expressed in AQL+.

use these extensions to refer to the logical variables and operators in the incoming logical plan during the optimization process, as the AQL+ transformation of a given plan happens during the optimization process. Note that the optimizer sees only the logical plan and physical plan, not the original query. Since AQL itself does not have an explicit join clause, AQL+ adds one in order to express a join of two branches. 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 the inputs of the AQL query and to refer to logical constructs that cannot be directly specified in AQL, such as operators in the plan. In this way, any AQL+ template can be combined with any join input branches, where the inputs can be from any kinds of subplans made up of other algebraic operators. In addition, to support various types of data, similarity functions, and thresholds, the similarity-join rule template uses placeholders that are parts of the AQL+ query and are unknown until runtime. These are used for data types, similarity-specific functions, or values. For example, a `SIMILARITY` placeholder is used for built-in AQL

```
select element {"rid":o.id, "iid":i.id}
from AmazonReview o, Reddit i
where similarity_jaccard(word_tokens(o.summary),
      word_tokens(i.title)) >= 0.8
```

(a) A simple similarity-join query



Lines* indicates the lines in the AQL+ template where the given AQL+ constructs are used.

(b) A logical plan for the query and AQL+ constructs.

Fig. 22. An example query and the corresponding logical plan that AQL+ template receives.

functions, and a `THRESHOLD` placeholder is used for numerical similarity values.

Specifically, for the three-stage-similarity join, the optimizer 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 this information, the logical plan fed into the AQL+ template can be transformed into the equivalent three-stage-similarity-join plan. Again, rather than doing this transformation by introducing operators by hand, we rely on the existing compilation path to generate a revised plan. This process is depicted in Fig. 20; the details of this optimization flow will be discussed in the next subsection. (See Table 1.)

The optimizer uses the AQL+ three-stage-similarity-join query template shown in Fig. 21 to transform the incoming user query during the rule-rewrite phase. In this way, the simple user-written query of Fig. 6(a) can be transformed into the query of Fig. 15 during the optimization process. The details of this AQL+ template are as follows. (We mostly focus on the AQL+ constructs here.)

Stage 1: *Token Ordering*. This stage is expressed in lines 14–21 of Fig. 21. The first meta-clause `##RIGHT_3` refers to the left input operator of the given join. Since the same branch can be used several times in each stage of the three-stage-similarity-join, if there are dependencies between the reused branches, deep copies of the given branch will be created. The suffix _3 here denotes that this is the third copy of the given branch. In the next line, `$id` is assigned to the primary key of the left branch, which is denoted by a meta-variable, `$$RIGHTPK_3`. The suffix _3 means the third copy of the branch as described. `TOKENIZER` is a template placeholder that will be replaced by an actual tokenizer function. For example, the string tokenizer will be used for a string field. Note that if `$$RIGHT_3` is an array or a multiset, no tokenizer will be added.

Stage 2: *Record ID (RID)-Pair Generation*. This stage is expressed in lines 6–51. This stage starts with a `join` meta-clause. This meta-clause has three arguments, and each argument is separated by a comma. The first argument is the left input branch of the join. The second argument is the right input branch of the join. The third argument describes the join condition. Since AQL does not have an explicit join clause, the `join` meta-clause in AQL+ provides a join expression that can be directly translated into a logical JOIN operator. In this join, the left branch is expressed in lines 8–27. Lines 29–50 denote the right branch of the `join` meta-clause. Line 51 of the template denotes the join condition itself. Specifically, the left branch starts with `##RIGHT_1`, which means the first copy of the branch. In the next line, tokens will be generated from the right variable of the original JOIN operator. In line 25, the prefix tokens for the right branch are calculated. Here, `PREFIX_LEN` denotes the function that calculates the prefix length based on the similarity type and the threshold. `THRESHOLD` contains the similarity threshold. In line 46, the prefix length for the left side is calculated. This calculation is different from that of the right branch, as a token that is in the right branch may not exist in the left branch. This calculation is needed since the template will be applied for both *R-S* joins and self-joins (*R-R*). The join condition is in line 51.

Stage 3: *Record Join*. This stage, which fetches the fields for similar records for the final result, is expressed in lines 1–5 and 56–57, each of which consists of two `join` meta-clauses. The first `join` meta-clause adds the record information for the right branch and the second `join` meta-clause adds the record information for the left branch. The conditions for the two joins are in lines 56–57.

To apply this AQL+ template to transform a nested-loop-similarity-join plan into the three-stage-similarity-join plan, some preparation must be done. All meta-variables, meta-clauses, and placeholders need to be replaced by actual operators, variables, and logical JOIN operators in the three-stage-similarity-join optimization rule. For example, in the AQL+ template, the primary key of each branch are referred using `##RIGHTPK` and `##LEFTPK` variables. These meta-variables are replaced with the actual primary keys in that optimizer rule. Fig. 22 shows an example similarity-join query and its logical plan including the AQL+

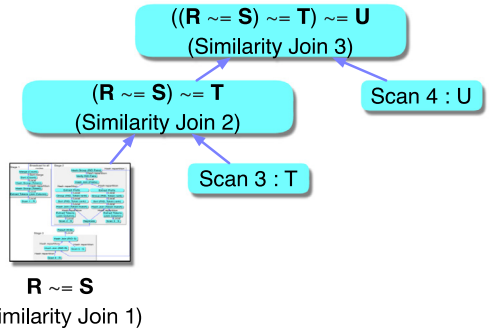


Fig. 23. Rewriting a multi-way-similarity-join plan on four datasets.

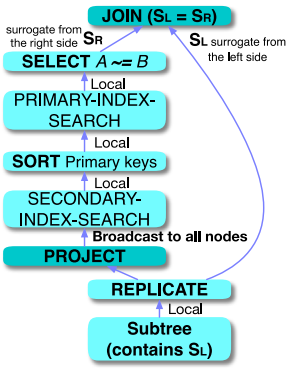


Fig. 24. Surrogate index-nested-loop-join plan.

constructs used in the AQL+ template. We can see that the top-most operators in both join branches form meta-clauses. Also, the primary keys from the datasets in both branches are regarded as meta-variables. The variables used in the similarity-join condition are also meta-variables in the plan. These AQL+ constructs appear in the AQL+ template in Fig. 21. For example, the ##LEFT meta-clause is used in line 3.

In addition to two-way similarity joins, the AQL+ framework can be applied to transform multi-way-similarity join plans as well because the optimizer can transform a logical plan iteratively. Similar to non-similarity-join cases, multi-way-similarity joins can be transformed sequentially. For instance, Fig. 23 shows a similarity-join plan involving four datasets. The join between the 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*.

It is worth noting that AQL+ is a general extension framework, not only for similarity queries, that in principle can be used to support other transformations expressed via AQL during the compilation process. (This was part of the original vision for AQL+.)

5.3. The optimization rule for similarity queries

As discussed in Section 2.3.1, the optimization process in AsterixDB is rule-based. Once the Algebricks layer receives a compiled plan from a SQL++/AQL query, it optimizes the plan both logically and physically. It first optimizes the given plan logically using several rule sets.

To apply the similarity-query optimization framework to the current optimization path, we created a new rule set for the AQL+ framework and similarity queries as was shown in Fig. 20. 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 a JOIN operator. If its condition includes a similarity predicate, it applies the AQL+ template to the plan to generate an AQL+ query. It then compiles the query into a new logical Algebricks plan. During this process, all meta-variables, meta-clauses, and placeholders are replaced by actual variables, operators, and logical JOIN operators. At this point, some parts of the overall query plan will have already been optimized if they belonged to the original incoming plan. However, most parts of the plan will not have been optimized yet, as the three-phase plan has just been compiled and has not gone through the optimization process before the application of the SJR rule set. Therefore, the newly generated plan needs to go through some of the earlier optimization rules again to ensure that those rules have a chance to process all of the newly added plan fragment's constructs. Note that this re-application process is not necessary for non-similarity queries since the plan generated for a non-similarity query is not touched by the SJR rule set. Therefore, we ensure that the similarity-join rule set is only applied to similarity-join queries. A benefit of this approach is that the optimization steps for similarity queries can be executed without interfering with those for non-similarity queries; this approach also gives the newly generated similarity-query plan a chance to reach the same level of transformation once the similarity rule set has finished its work.

5.4. Improvements

We now discuss two improvements that we employ in similarity query processing that could be applied to general query processing as well.

5.4.1. Surrogate index-nested-loop-join

A general drawback of an index-nested-loop join using a local secondary index is the need to broadcast the outer side's data to all secondary-index partitions, as explained in Section 4. For example, during the execution of the query in Fig. 12, the outer side needs to broadcast the join key field `summary` as well as the `reviewer_id` and `id` fields. If there were more fields in the return clause, the broadcast cost would increase as well. This broadcast step is a consequence of the co-partitioning of each secondary index with its primary index. Another issue is that a secondary-inverted-index search can generate multiple pairs of results for the same primary key, as there can be multiple entries of secondary keys for the same primary key. Thus, we would like to reduce the cost of the sorting step between the secondary-index search and primary-index search. We reduce this cost by only sending the similarity-related secondary-key fields together with a compact surrogate for each outer-side record, and then later we use the surrogates to obtain the surviving original records. This approach is reminiscent of semi-join optimization in distributed databases [45]. In AsterixDB, we use the primary key of a dataset as a surrogate.

Fig. 24 shows a surrogate-based index-nested-loop-similarity join plan. Notice the PROJECT operator that follows the REPLICATE operator after the outer subtree; it eliminates all non-essential fields from the outer side. The optimizer filters out the `reviewer_id` field since the search key is the `summary` field and the primary key is the `id` field. In addition, since the same subtree is used twice in the plan, a REPLICATE operator is introduced to

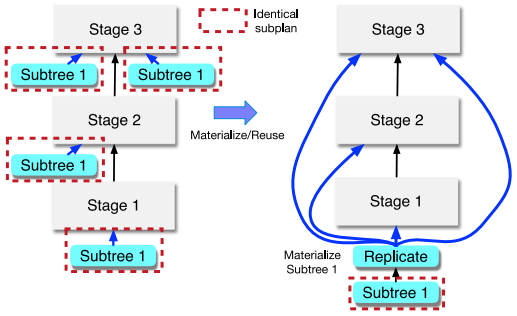


Fig. 25. Reusing a subtree of a self three-stage-similarity join.

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 associated 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 equijoin on the surrogates S_L and S_R, it can be executed efficiently in parallel, e.g., using a hash join. This surrogate-based join optimization is currently always used in the three-stage-similarity join and inverted-index-nested-loop join. It could also be used for other index-nested-loop joins (but currently it is not).

5.4.2. Materializing/reusing shared subplans

As shown in the simplified sketch of a three-stage-similarity join in the left part of Fig. 25, in case of a three-stage-similarity self-join, the dataset R needs to be scanned four times. We could simply execute the original data-scan operation four times if R was just a full stored dataset. However, if the branches of this join result from a complex computation from a subquery, it would be expensive to recompute the result of the subquery multiple times. To minimize this cost, AsterixDB instead materializes the common subplan and reuses it several times. This is done by computing the common branch once, materializing the results if necessary, and replicating and pushing the results to all the operators that have the root of one of these isomorphic subgraphs as an input using a REPLICATE operator as shown in the right part of Fig. 25.

To implement this optimization as a general rule that can be applied to any logical plan, we first need to identify whether a certain part of the plan is provided as an input to more than one operator. We find equivalence classes where each equivalence class includes a set of isomorphic subgraphs.

After identifying the original subgraph and its instances in an equivalence class, we need to decide whether the results of the original subgraph need to be materialized or not. In some cases, the results of the original subgraph can be pipelined through the REPLICATE operator and the operators after the REPLICATE operator can access them at the same time. In this case, the results do not have to be materialized. However, due to possible wait-for dependencies among the operators that have the original subgraph as an input, it is not always possible to run these operators at the same time. For example, consider a case where a common subgraph G is fed into two operators A and B and A cannot start until B finishes because of wait-for dependency between A and B. In this case, the result of G cannot be pipelined to A and B at the same time. We then materialize the results to a temporary file and read from it when processing A. The decision about materializing an input before replicating it depends on how the operators can be co-scheduled.

Table 2
AsterixDB parameters for the experiments.

Parameter	Value
Global memory budget per node	6 GB
Budget for in-memory components	3 GB
Data page size	128 KB
Disk buffer cache size	2 GB
Sort buffer size	128 MB
Join buffer size	128 MB
Group-by buffer size	128 MB

To identify the wait-for dependencies among operators for each identified equivalence class to determine whether materializing the subgraph is required, we check the activities of each operator in the plan and the wait-for dependencies between these activities, since the basic execution unit in Hyracks is an activity [43]. Using this wait-for dependency information, we group operators that can be co-scheduled as an activity cluster and assign an id to it. If there are no wait-for dependencies among the activities of an operator, the operator can be put into a group with its previous operator(s) and its next operator(s). However, if there are wait-for dependencies among the activities of an operator, it is not possible to place this operator with the previous operator and the next operator in a group. For example, a SORT operator has two activities. The first activity builds temporary run files and the second activity merges these files to generate the results. The second activity depends on the first activity since it cannot start until the first activity finishes. If a SORT operator is placed between an operator A and an operator C, the operator C cannot be co-scheduled with the operator A because of this dependency. Based on the activity cluster information, we then construct a wait-for dependency graph among the clusters. For each instance of the identified subgraph in an equivalence class, we use the cluster id that contains the root of the instance as the representative id and look for wait-for dependencies between these representative ids in this graph. If a representative id is blocked by one or more representative ids, the input for the operator that has the instance of the identified subgraph in the cluster should be materialized. Otherwise, the input for the operator will not be materialized.

6. Experiments

We have conducted an experimental evaluation of our approach in AsterixDB using large, real datasets. We used an 8-node cluster to host an AsterixDB (0.9.3) instance, where each node ran Ubuntu with a Quadcore AMD Opteron CPU 2212 HE (2.0 GHz), 8 GB RAM, 1 GB Ethernet NIC, and had two 7200 RPM SATA hard drives. Each dataset was horizontally partitioned into 16 partitions (2 per node) based on their primary keys to provide full I/O parallelism. Table 2 shows the AsterixDB configuration parameters.

6.1. Datasets

We used several similarity functions to experiment with 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 with different similarity functions, we used the three datasets with different characteristics shown in Table 3. The `Amazon Review` dataset, discussed in earlier sections, included Amazon product reviews from [46]. The `Reddit Submission` dataset contained about eight years of postings on Reddit from [47]. The `Twitter` [48] dataset had 1% of US tweets for three months that we obtained ourselves via Twitter's public API. When imported into AsterixDB, each dataset

Table 3
Dataset characteristics.

Dataset	Amazon Review [46]	Reddit [47]	Twitter [48]
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 in AsterixDB	60.6 GB	305 GB	582 GB
Fields used	summary, reviewerName	title, author	text, user.name

Table 4
Characteristics of the search 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

Table 5
Index size and build time for Amazon Review dataset.

Field	Index type	Size (GB)	Build time (s)
Dataset itself	B+ tree (primary)	60.6	1563
reviewerName	B+ tree (secondary)	2.7	223
reviewerName	2-gram (secondary)	15.6	1441
summary	B+ tree (secondary)	3.5	275
summary	keyword (secondary)	5.4	573

Table 6
Candidate size and the final result size for the indexed-Jaccard-selection query for Amazon Review dataset in Fig. 27.

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%

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 pre-declared test schemas. 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 stored record contained additional information about each non-pre-declared field such as the field name, field type, and value. For example, for a string field named `summary`, each instance of the field `summary` will contain the field name `summary`, its type as string, and its value. (In contrast, if the field was explicitly defined in the schema, the field name and its type would not be required to be stored in each record.)

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

6.2. Index size

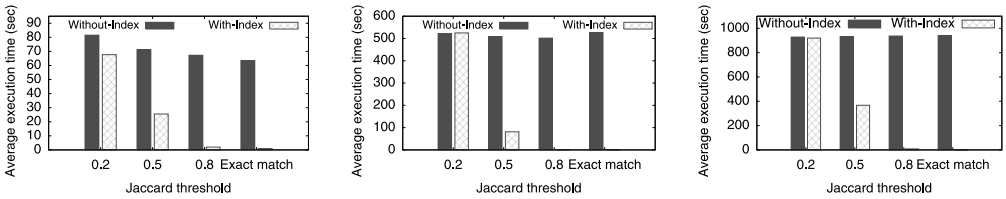
We built a keyword index for Jaccard similarity queries and a 2-gram index for edit distance queries. To measure the execution time for basic exact-match queries on the same fields to serve as a baseline, we also built a B+ tree index on each of the search fields. Table 5 shows the index sizes for the Amazon Review dataset and the time to create each index. An *n*-gram index took much more space than a B+ tree index or a keyword index, as it had more secondary keys per record. For instance, a 2-gram index on the `reviewerName` field needed 15.6 GB 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 it had multiple secondary keys per record. For each type of index, the construction time was roughly proportional to the size of the index. In each case, the dataset itself was also stored in a primary B+ tree index.

```
select value count(*) from (
  select ar.id
  from AmazonReview as ar
  where similarity_jaccard(ar.summary,
        "should have tried them at the store") >= 0.5
) as first_select;
```

Fig. 26. An example SQL++ similarity-selection query.

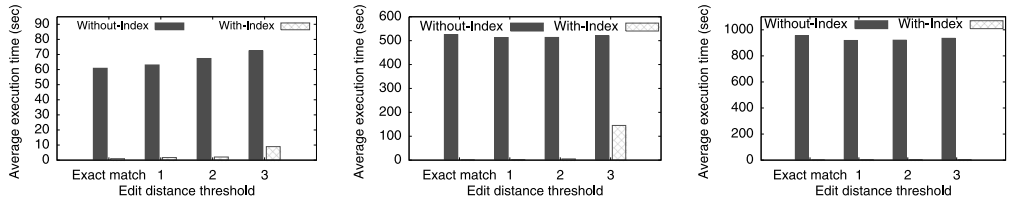
6.3. Selection queries

To measure the performance of similarity-selection queries, we first created a search value set that contained 10,000 random unique values that we extracted from the search field. For Jaccard queries, we ensured that the minimum number of words in each value in the search set was 3. For edit distance queries, the minimum length of characters in each value was 3. For each similarity threshold, we randomly chose search values from the set for a query and sent 100 such queries to the cluster, and measured their average execution time. The performance baseline for comparison purposes was an equality-condition query that used the same values for the given field. Fig. 26 shows an example query that we used to measure the average execution time of the Jaccard similarity queries. In this example, we used `similarity_jaccard` as the similarity function on the `summary` field



(a) Amazon Review (b) Reddit (c) Twitter

Fig. 27. Execution time of Jaccard selection queries on the three datasets.



(a) Amazon Review (b) Reddit (c) Twitter

Fig. 28. Execution time of edit-distance selection queries on the three datasets.

```
select value count(*) from (
  select element {"oid":o.id, "iid":i.id}
  from AmazonReview as o, AmazonReview as i
  where similarity_jaccard(o.summary, i.summary)
        >= 0.8
  and o.product_id = "B00103DCIZ" and o.id < i.id
) as first_join;
```

Fig. 29. An example SQL++ similarity-join query.

with the threshold of 0.5. The second parameter of the function was a random value from the above search value set.

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. Fig. 27 shows the results. We see that the average execution time for similarity selection queries decreased as the threshold increased in the case of index-based plans. For example, it took the index-based method 67.6 s to conduct a Jaccard query with a threshold of 0.2, while it took only 25.5 s to execute a query with a threshold of 0.5 on the Amazon Review dataset. 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 the data 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 took more time to calculate a Jaccard value than to get the result of an exact match. This overhead decreased as the threshold increased because we applied optimizations such as early termination and pruning based on string lengths, which significantly reduced the cost of computing the similarity. The trend is similar in the other two datasets.

When the threshold was low, the execution 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 large when the threshold was low, as shown in Table 6. As the number of candidates increased, the search time increased due to the need for a primary-index lookup and a verification for each candidate.

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. Fig. 28 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, as can be seen in Table 7. It took the index-based method 2 s to run a selection query with a threshold of 2; it took 8.9 s 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 first ran similarity-self-join queries on the three datasets. Fig. 29 shows an example query that was used to measure the average execution time as in the similarity-selection query case. Here, `summary` is the field on which we applied a similarity function and `id` is the primary key field. After conducting the self-join experiments, we conducted an additional multi-way join experiment that included both similarity and non-similarity joins. Finally, we conducted a multi-way similarity join experiment that had two similarity joins involving all three datasets in one query.

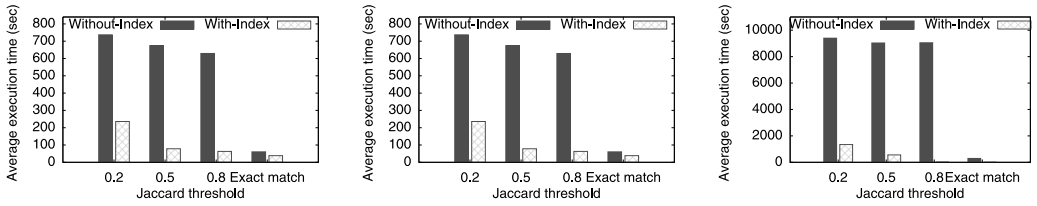
6.4.1. Varying threshold

We first extracted a certain number of records from the outer branch of the join to limit the size of its input. For each query, we chose 10 random records from the outer branch. In the example query in Fig. 29, the field named `product_id` was used to impose this limit. For Jaccard join queries, we used three similarity thresholds, namely 0.2, 0.5, and 0.8. For edit distance, we used distance thresholds of 1, 2, and 3.

The results are shown in Figs. 30 and 31. When there was no applicable index, AsterixDB used the three-stage-similarity-join plan for the Jaccard queries. The trends were similar to those of selection queries except for the exact-match join, which significantly outperformed both the Jaccard and edit distance joins since it used a hash join in which the join keys were broadcast to multiple nodes. For the index-nested-loop join case, all three datasets showed a similar trend on both the Jaccard

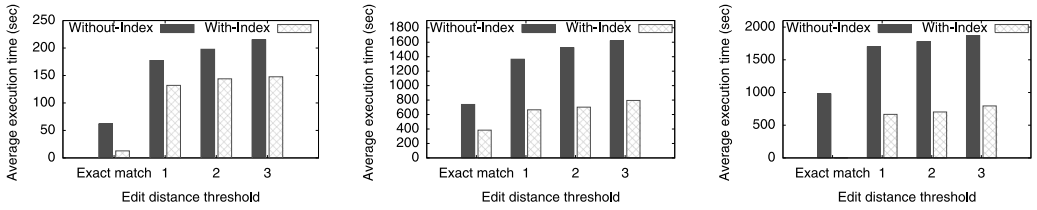
Table 7
Candidate size and the final result size for the indexed-edit-distance-selection query for Amazon Review dataset in Fig. 28.

Edit distance threshold	Actual result record count (B)	Candidate set record count (C)	Ratio (B/C)
1	52	64	81.25%
2	297	3,477	8.54%
3	4,185	239,166	1.75%



(a) Amazon Review (b) Reddit (c) Twitter

Fig. 30. Execution time of Jaccard join queries on the three datasets.



(a) Amazon Review (b) Reddit (c) Twitter

Fig. 31. Execution time of edit distance join queries on the three datasets.

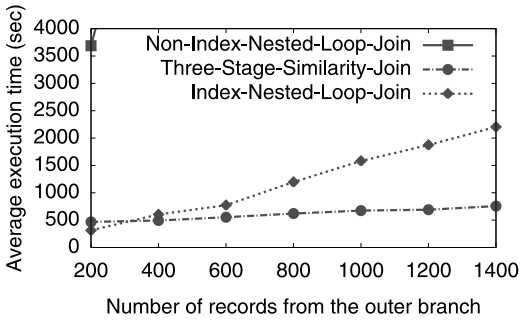


Fig. 32. Similarity joins on the Amazon Review dataset.

```
select value count(*) from (
  select element {"oid":o.id, "iid":i.id} from
  from ProductID as pr, AmazonReview as o,
       AmazonReview as i
  where pr.product_id = "B00103DCIZ"
  and pr.product_id = o.product_id
  and similarity_jaccard(o.summary, i.summary) >= 0.8
  and edit_distance(o.reviewerName, i.reviewerName)
      <= 1 and o.id < i.id
) as first_join;
```

Fig. 33. An example SQL++ multi-way-join query.

and edit distance joins. For instance, for the Jaccard queries, as the threshold increased, the average execution time decreased as well.

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 of Fig. 30 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, which was small relative to the time required to actually execute the resulting query plan. (See Fig. 31.)

6.4.2. Varying input size

To further explore the relative performance of the join methods, we conducted a Jaccard join with a fixed threshold and varied the number of records to be joined. 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 coming from the outer 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 Fig. 32. 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 the highest by far at 200 records and that it increased drastically compared to the 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 its secondary-index search, as it deals with each record one at a time. For the three-stage-similarity join, the time spent on global-token-order generation in the first stage was the same for all cases, since the order was generated from the inner branch and we only varied the number of records from the outer branch. The hash joins utilized in stage 2 and 3 can deal with the incoming records efficiently since each join key (a token) is sent to only one node. In fact, the average execution time increased slightly for the three-stage-similarity-join case as the number of records that need to be processed in stage 2 and 3 was

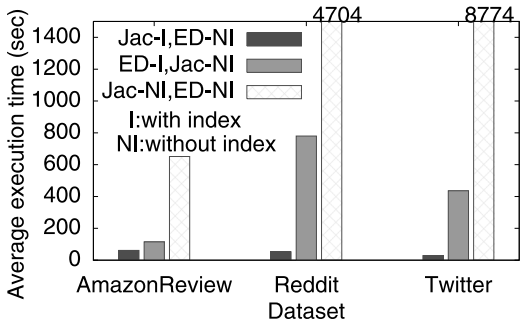


Fig. 34. Multi-way-join queries on the three datasets.

increased as well. This slight increase of the average execution time of the three-stage-similarity join can be verified in the figure. For instance, the time for the three-stage-similarity join for 800 records was 619 s, while it was 674 s for 1000 records. This result shows only 55 s of increase, whereas the execution-time difference for index-nested-loop joins when going from 800 to 1000 input records was 384 s.

6.4.3. Multi-way join queries

So far we have used only one similarity condition per query. Next, we added one more similarity condition to the query and varied the order of the conditions. A similarity join is conducted with the first condition and then a SELECT operator with the other predicate is applied after the join. These similarity conditions were a Jaccard condition with a threshold of 0.8 and an edit distance condition with a threshold of 1. We also added an initial equijoin to control the number of records being fed into the similarity join. This join is applied first to generate a fixed number of records. Fig. 33 shows an example query that we used to measure the average execution time. As we see in this query, there is one similarity join and one equijoin. The dataset `ProductID` and the field `product_id` were what we used to limit the number of initial records from the outer branch.

For the first equijoin, we used an index-nested-loop join to fetch the initial records quickly to avoid a full-scan of the dataset. The Jaccard similarity and edit distance conditions were then applied. In the cases where we applied the Jaccard condition first, the Jaccard join was followed by the edit distance condition in a SELECT operator. For both conditions, we used an index-based method for the first and a non-index-based method for the second. That is, we tried three types of queries in total. The first query initially used the indexed Jaccard similarity join. The second query used the indexed-edit distance join first. The last query used the non-indexed Jaccard join first. Fig. 34 shows that the performance was the best when the index-based-Jaccard join was conducted first, as then there were no corner cases for Jaccard similarity. This similarity predicate order also generated fewer candidates than applying the index-based edit distance predicate first. In contrast, for the edit distance case, the optimizer needed to augment the corner-case path in the logical plan, and thus it generated more candidates.

6.4.4. Multi-way three-stage-similarity join queries

The previous join experiment used a non-similarity-index-nested-loop join and a similarity join. After two joins, a second similarity predicate was applied via a SELECT operator. To test the performance of a query with multiple three-stage-similarity-joins, next we used all three datasets in one query as shown in Fig. 35. First, we fetched ten random records from the Amazon Review dataset and conducted a three-stage-similarity-join between the `summary` field of the dataset and the `title` field of

```
select value count(*) from (
  select element {"tid":tw.id, "sid":second.sid,
                  "fid":second.fid}
  from Twitter tw, (
    select element {"sid":re.id, "fid":first.oid,
                    "title": re.title}
    from Reddit re,
      (select element {"oid":o.id, "summary":o.summary}
       from ProductID as pr, AmazonReview as o
       where pr.product_id /* +indexnl */ = o.product_id
       and pr.id = "B00103DCIZ"
      ) as first
    where similarity_jaccard(re.title, first.summary)
          >= 0.8
  ) as second
  where similarity_jaccard(tw.text, second.title)
        >= 0.8
) as third;
```

Fig. 35. An example SQL++ multi-way three-similarity-join query.

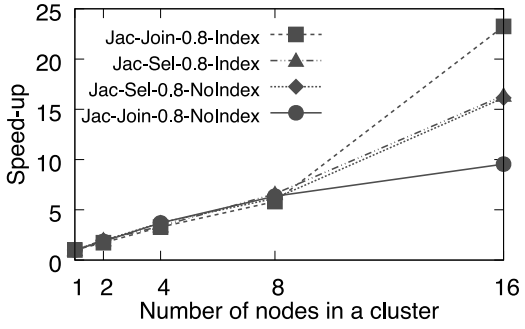


Fig. 36. Speed-up on Jaccard on Amazon Review dataset.

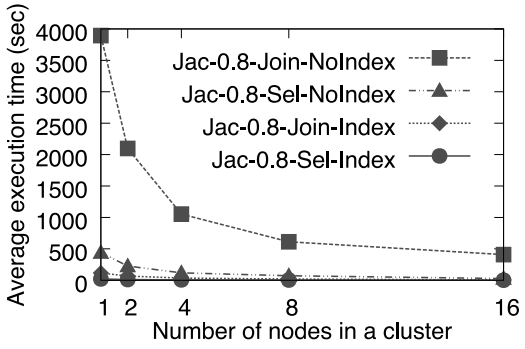


Fig. 37. Times for Jaccard speed-up on Amazon Review dataset.

the Reddit dataset. The result was used to conduct a join with the `text` field of the Twitter dataset. We ran this multi-way join query three times with ten different records each time. The resulting average execution time was 6908 s and the average result count was 737,406. Note that we used the Reddit and Twitter datasets as the outer branches of two three-stage-similarity-joins since the global token order was generated from the inner branch that fetched ten records from the Amazon Review dataset. An example record that this query found was "So `Comfy`", "So `comfy....`", and "So `comfy`" from the Amazon Review, Reddit, and Twitter datasets respectively.

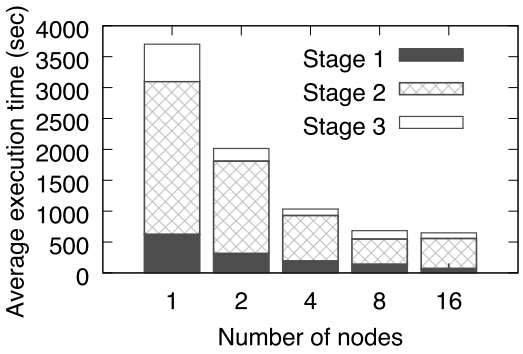


Fig. 38. Per-stage execution time of the three-stage-similarity-join query on Amazon Review dataset.

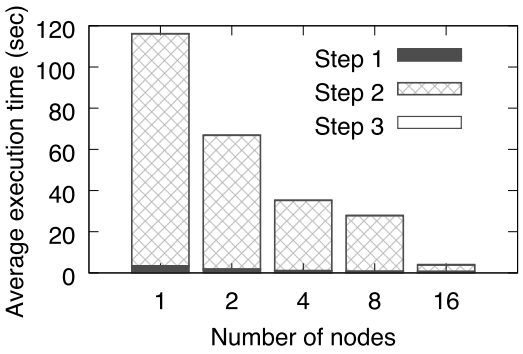


Fig. 39. Detailed execution time of index-nested-loop-Jaccard-join queries on Amazon Review dataset.

6.5. Cluster scalability tests

We used both speed-up and scale-out metrics to evaluate similarity-query processing in a parallel environment.

6.5.1. Speed-up

First, for our speed-up experiment, we used five cluster sizes (1, 2, 4, 8, and 16 nodes), with each cluster size being given the entire (100%) dataset to spread out across its partitions. Figs. 36 and 37 show the speed-up and average execution time of the previous Jaccard selection and join queries with the threshold set to be 0.8. For each type of query, we measured the average execution time of 100 indexed and 100 non-indexed queries. The speed-up of the selection queries was proportional to the number of nodes. However, both of the join queries showed non-linear behaviors here that we did not observe earlier in [5], where the maximum number of nodes was 8. With 16 nodes, the graph here showed a clear distinction among these queries. (Another difference here is that we increased the number of queries from 10 to 100.)

One observation is that the speed-up of the three-stage-similarity-join query in Figs. 36 and 37 appears to be sub-linear. This is mainly due to the communication cost among all nodes, as the three-stage-similarity-join involves a number of tuple exchanges (as was shown in Fig. 16). In particular, before each hash join, every tuple from both join branches is hash-partitioned to a potentially different node based on the join key's hash value. Except for the joins in stage 3, where extra fields from two branches of the original similarity join are being fetched, all hash joins are conducted on a token or a prefix (based on the global token order). That is, each field in the original query generates more join keys that are joined in stage 2 since each field value is tokenized. The tokens from both branches in these joins need to be hash-partitioned. Also, in stage 1, after the sorting on the token frequency on each node is done, the partial results from each node need to be merged on one node to create the global token order. This merge operation is conducted in a serial fashion; thus, it became a bottleneck for the global sort operation. Based on these characteristics, the three-stage-similarity-join can be regarded as a communication-bound process and the "sub-linear" behavior of the three-stage-similarity-join stems from the fact that the speed-up of a heavily communication-bound process is about $\frac{k}{2}$, where k is the number of nodes, as explained in Appendix A. If there is only a small number of nodes in a cluster, the speed-up is less than $\frac{k}{2}$. As we increase the number of nodes, we see the eventual linear nature of the speed-up in the graphs, which indeed goes as approximately $\frac{k}{2}$. Although $\frac{k}{2}$ is a linear speed-up trend, still, the ratio is less than k.

Fig. 38 shows the per-stage-execution-time of the three-stage-similarity-join query on each cluster setting. We can see that the most time was spent on stage 2. Most of the communication cost in stage 2 is due to the fact that we need to tokenize the given field from the dataset and conduct a hash join to match each token against each token in the global token order chosen in stage 1. In fact, we observed that on the 16-node cluster, it took 255 s to exchange the tokens and the primary keys among all the nodes in one of the hash-joins of stage 2. Since the query took 423 s in total, 60% of the query execution time was spent just on the hash-partition exchange involved in stage 2.

In contrast, the speed-up of the indexed-nested-Jaccard-join query in Fig. 36 is seen to be super-linear. To explain this behavior, we can decompose the indexed-nested-Jaccard-join query into three steps. In step 1, AsterixDB extracts 10 random tuples from the outer branch and broadcasts them to all nodes. It then extracts the given field, tokenizes the field value, and conducts keyword-index searches using the tokens. The keyword-index search yields candidate primary keys. In step 2, these primary keys are sorted and fed into the primary-index lookup. AsterixDB extracts the field from the record to again verify the Jaccard condition and other predicates. In step 3, AsterixDB employs a surrogate hash-join at the top level to merge any other fields that are not the secondary key or the primary key fields since this is an inverted-index-join. After this join, the count of primary keys will be gathered and returned to the user. Since the surrogate-hash-join is for primary keys on the same dataset, there is little communication required since records are partitioned on the primary key. Fig. 39 shows the execution time of this join query per stage. Most of the time was spent in step 2, as shown in the figure. Note that the speed-up of each operation in each step is linear except for the sorting operation. That is, when we reduce the size of the dataset partition on each node, the amount of work is reduced linearly. For example, if it takes 1 s to conduct 1000 primary key lookups on a 1-node cluster, it takes about 0.5 s to conduct 500 primary key lookups on each node of a 2-node cluster. Unlike other operations in the query plan, The speed-up of the sorting operation is $k \cdot \log_{\frac{N}{k}} N$, where N is the number of tuples and k is the number of nodes, as explained in Appendix B. This ratio is super-linear because of its second term, which explains the super-linear behavior of the query. For instance, on a 16-node cluster, the speed-up was 20.02 (>16) when N was 1 million.

One more observation was that the speed-up on the 8-node cluster was lower than expected in general because there was skewness of the data distribution among the nodes in the cluster. In our experiment, when loading the data, each record received a randomly generated UUID value as its primary key. Therefore, in each cluster setting, the actual data distribution was different.

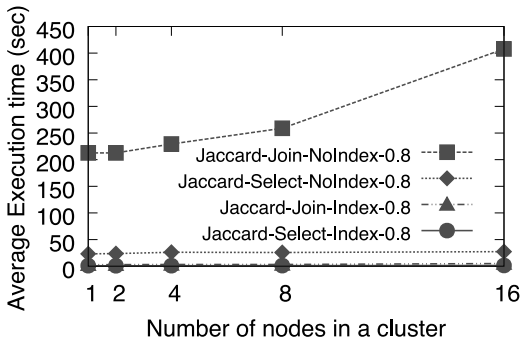


Fig. 40. Scale-out for Jaccard on Amazon Review dataset.

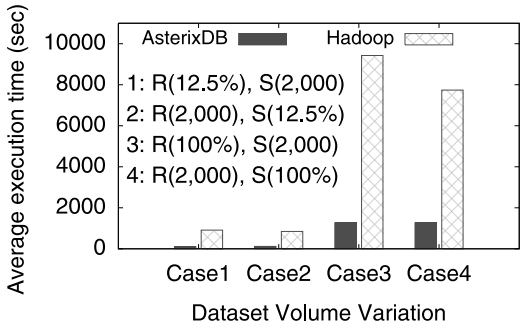


Fig. 41. Three-stage-similarity-join queries on AsterixDB and Hadoop Map/Reduce.

Note that our search was conducted on a secondary key field, not the primary key. We observed that on the 8-node cluster, it took about 27 s to conduct step 2, and there was a 17-second difference between the time when the first node finished and the time when the last node finished this step. This showed the skewness of the data distribution.

6.5.2. Scale-out

To explore scale-out, we again used five clusters of different sizes, namely 1, 2, 4, 8, and 16 nodes. In this case, however, when we doubled the number of nodes in the cluster, we also doubled the data size to yield the same amount of data per node. The 1-node cluster had just 6.25% of the original total dataset size, the 2-node cluster had 12.5% of the data, the 4-node cluster had 25% of the data, and the 8-node cluster had 50% of the data. The 16-node cluster had the entire original dataset. Ideally, for linear scale-out, the response-time graph would show a flat line per query. In fact, the response times for each cluster size were similar, as shown in Fig. 40, except in the case of the ad hoc Jaccard-similarity join without an index. As we described in the speed-up section, this was due to the fact that the three-stage-similarity-join is a communication-bound join method, so its communication cost increases as the number of the nodes increases in a cluster based on the fact that the volume of data on each node remains the same.

6.6. Comparison with other systems

In addition to evaluating the performance characteristics of AsterixDB's algorithms, we evaluated the performance of similarity queries on three other systems that also support certain types of similarity queries. They are basic Apache Hadoop, Couchbase, and Elasticsearch. We selected these three systems because we had previously implemented the three-stage-similarity-join as a map/reduce job in Apache Hadoop and because Couchbase and Elasticsearch support scalable edit distance queries.

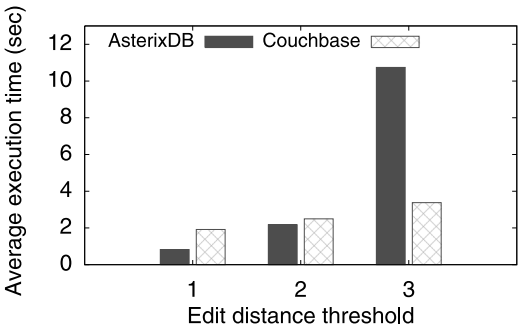


Fig. 42. Edit-distance queries on AsterixDB and Couchbase.

6.6.1. Apache Hadoop

The three-stage-similarity join [4] was originally implemented by hand using Apache Hadoop. Based on the original code [49], we replicated the three-stage-join experiment on Apache Hadoop Map/Reduce 1.2.1, the most recent version compatible with that three-stage-similarity-join code. To make the execution environment similar to that of AsterixDB, we designated one node to host master daemons to run the Hadoop jobs and to control the Hadoop Distributed File System (HDFS). Including this node, eight nodes were utilized to run map and reduce tasks. Each node had two directories since there were two AsterixDB partitions. We set the HDFS block size to 128 MB and allocated 1 GB of virtual memory to each HDFS daemon. Since AsterixDB used 2 GB as buffer space, we allocated 2 GB of virtual memory to each map/reduce task. We ran two map tasks and two reduce tasks on each node so that the degree of parallelism was also the same for both systems. The replication factor was set to 1, and Hadoop's speculative task execution feature was disabled. Fig. 41 shows the execution times for the same three-stage-similarity-join query that used Jaccard with a threshold of 0.8 for several different cases (explained below).

One difference between the Apache Hadoop implementation and AsterixDB is their global token order generation in stage 1. When calculating the global token order of an R and S join, Hadoop uses R to build the global token order. AsterixDB instead uses S to build the global token order. Thus, we experimented with four variations for the dataset size in Fig. 41. In the first case, the left branch (R) used 12.5% of the tuple of the Amazon Review dataset while the right branch (S) used 2000 tuples of the same dataset. We then switched the left and the right branches in the second case. By checking these two cases in Fig. 41, we can see that the execution time was smaller when the size of the dataset used to generate the global token order was smaller. We also see that the execution time for AsterixDB was about ten times faster than that of Hadoop. This was because the execution of AsterixDB is pipelined whenever possible. In contrast, the result of each map-reduce task is written to disk and then read again in Hadoop.

6.6.2. Couchbase

As described in Section 1.1, Couchbase is unique in providing support for edit distance search queries on NoSQL data with its new full-text search service. It does so via a separate full-text API (not its N1QL query language). A full-text index must be built before sending full-text queries involving edit distance. Given the provided support, only an indexed-edit-distance query comparison between AsterixDB and Couchbase is appropriate. In our experiments, AsterixDB had two physical partitions on two separate hard disks on each node to increase its degree of parallelism. Since Couchbase can only support multiple physical partitions using a redundant array of independent disks (RAID),

Table 8
The execution time of the first query and the last query in Elasticsearch.

Threshold	First query (s)	Last query (s)
1	2.071	0.19
2	1.491	0.097

Table A.1
Communication cost per node on a hash exchange operation.

Number of nodes	Communication cost per node in terms of the original data	Speed-up
2	$1/2^2$	2
4	$3/4^2$	2.67
8	$7/8^2$	4.57
16	$15/16^2$	8.53
32	$31/32^2$	16.52
64	$63/64^2$	32.51
128	$127/128^2$	64.5
...		
k	$(k-1)/k^2$	$\approx k/2$

Table B.1
Per node cost of a parallel sort (1 million tuples).

Number of nodes	Sorting cost on a node	speed-up
1	$1,000,000 \cdot \log\ 1,000,000$	1
2	$500,000 \cdot \log\ 500,000$	2.11
4	$250,000 \cdot \log\ 250,000$	4.45
8	$125,000 \cdot \log\ 125,000$	9.42
16	$62,500 \cdot \log\ 62,500$	20.02
...		
k	$\frac{N}{k} \cdot \log\ \frac{N}{k}$	$k \cdot \log_{\frac{N}{k}}\ N$

we also ended up using only one partition on each node of the AsterixDB cluster for this comparison experiment. For Couchbase, we used version 5.0, and we set up the full-text service on the same nodes and allocated 2 GB of the memory to the full-text service on each node. After loading the Amazon Review dataset into both systems, we first sent a few dummy queries to warm up the instance. We then sent ten random indexed-edit-distance queries on the reviewer name field to AsterixDB and Couchbase and measured their average execution times. The results are shown in Fig. 42.

When the edit distance threshold was 1 or 2, AsterixDB performed better than Couchbase. When the threshold was 3, however, AsterixDB became about five times slower than Couchbase. A careful investigation revealed that the main reason is that the inverted-index search in AsterixDB generated many candidates as the threshold increased. These candidates needed to be verified via a primary-index search and applying the edit distance function on the fetched field. That is, an inverted-index search alone in AsterixDB cannot generate the final result, as described earlier. In contrast, the full-text index in Couchbase alone can generate the final answer without having to check the original data because their index contains all the data needed to generate the final answer. That is, a tentative result from an edit distance query is then verified within the full-text index to generate the final result. Note that this design implies that there may be inconsistencies between the full-text index and the actual data in a bucket until the synchronization between a bucket and the full-text index is done. In contrast, AsterixDB always generates an answer consistent with the most current data.

6.6.3. Elasticsearch

Elasticsearch is a distributed search engine for documents [50]. It utilizes a Lucene index to perform edit-distance searches. Since Elasticsearch does not support Jaccard similarity, only an indexed-edit-distance query comparison between AsterixDB and Elasticsearch is appropriate as in the Couchbase case. To be aligned with the Couchbase experiment setting, we used only one shard (partition) on each node of the AsterixDB cluster and the Elasticsearch cluster. We used Elasticsearch version 7.3 and allocated 6 GB of memory on each node. The number of replicas in the index was set to zero to be consistent with the setting of AsterixDB. After loading the Amazon Review dataset into both systems, we first sent a few dummy queries to warm up the instance. We then sent 10 random indexed-edit-distance queries on the reviewer name field to AsterixDB and Elasticsearch, and measured their average execution times. The results are shown in Fig. 43.

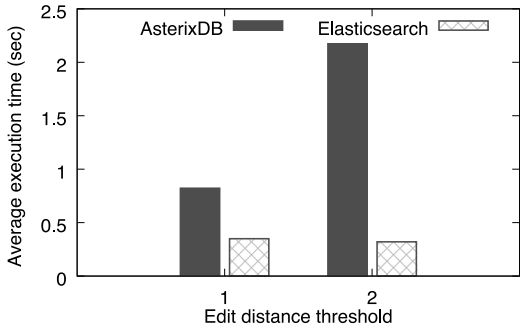
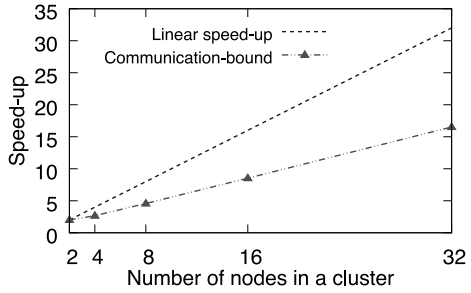
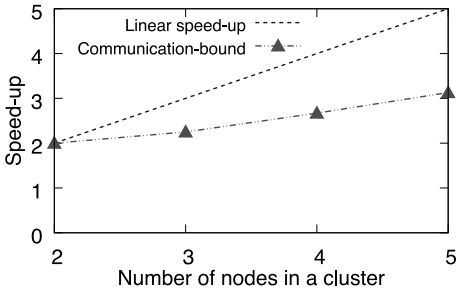


Fig. 43. Edit-distance queries on AsterixDB and Elasticsearch.

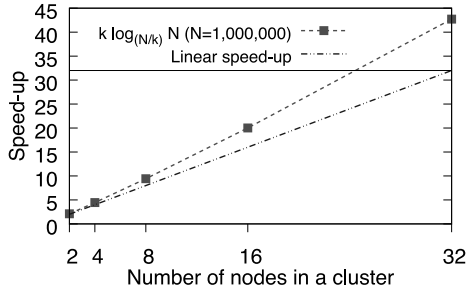
Here is an analysis of the results. Elasticsearch uses a Levenshtein automaton [51] to generate an answer, and supports an edit-distance threshold up to 2 due to performance considerations. Thus, we could not compare the case when the threshold was 3 or higher. For both thresholds 1 and 2, Elasticsearch performed better than AsterixDB. The main reason is that AsterixDB needs to go through its post-verification step, which involves accessing the data records, whereas this step is not needed in Elasticsearch (making its queries index-only in nature). That is, Elasticsearch constructs a deterministic finite automaton (DFA) that accepts all terms that are within the edit-distance threshold for the query's term and then iterates over those terms in the



(a) speed-up of a parallel communication-bound job (up to 64 nodes).



(b) speed-up of a parallel communication-bound job (up to 5 nodes).



(c) speed-up of a parallel sort job

Fig. A.1. Speed-up of communication-bound parallel job and parallel sort job.

index to generate results. AsterixDB first generates candidates from a secondary-index search. After that, it fetches the actual records to extract their field and calculates the real edit distance between the query keyword and the extracted field value. In addition, it has previously been shown that the performance of the N-gram-based approach adopted by AsterixDB is more suitable for long strings [52].

Notice that the execution time of the first query in Elasticsearch was ten times slower than that of the later queries as shown in Table 8. For instance, when the threshold was 2, the execution times of the first and last query were 1.491s and 0.1s, respectively. The average execution time of the queries without the first query was 0.15s. This is because the first query needed to load the index into memory. Once the index was loaded, the execution time decreased and finally became stable. This explains why the average execution time of the threshold-1 case was slightly longer than that of the threshold-2 case in Elasticsearch, as the time needed for loading an index into memory can fluctuate from time to time.

6.6.4. Evaluation

Considering these experimental results, when the edit distance threshold is relatively small, Elasticsearch outperformed AsterixDB and Couchbase. Elasticsearch showed a similar execution time for the two thresholds that it can support with its Levenshtein Automaton based implementation. However, it does not support thresholds larger than 2. For thresholds of 1 and 2, AsterixDB performed better than Couchbase. However, AsterixDB's performance degrades when the threshold becomes larger. In contrast, Couchbase showed a better-behaved (essentially linear) execution time degradation when the threshold increased.

7. Conclusions

In this paper, we have described an approach to providing integrated support for similarity queries in a parallel Big Data management system. We used Apache AsterixDB to illustrate and validate our approach. We described the entire lifecycle of a similarity query in the system, including the query language, indexing, execution plans, and plan rewriting to optimize query execution. Our similarity search solution leverages the existing infrastructure of a parallel data management system, including its operators, query engine, and rule-based optimizer. We presented an experimental study based on several large, real-world datasets on a parallel computing cluster to evaluate the proposed techniques and showed their efficacy and performance for supporting similarity queries on large datasets using parallel computing. To put our results in the broader context of parallel data platforms, we also presented and discussed a performance comparison with three other parallel systems (Hadoop, Couchbase, and Elasticsearch). We hope that others seeking to integrate search functionality into a general parallel data management context will find the results of our work to be useful.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The Apache AsterixDB project has been supported by an initial UC Discovery grant, USA, by NSF IIS, USA awards 0910989, 0910859, 0910820, 0844574, and by NSF CNS, USA 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.

Appendices

In this Appendix, we present detailed analyses of the speed-ups of an idealized communication-bound parallel job and of idealized parallel sort job to explain the super-linear and sub-linear speed-up behaviors of the two queries seen in Section 6.5.1.

Appendix A. Communication-bound parallel job

If a perfectly parallel data analysis job is totally communication-bound, how much speed-up can be expected? Table A.1 shows the fraction of the original data that must be transferred from each node in a single data-exchange operation, where k is the number of nodes. In this simple analysis, we assume that a data-exchange function evenly hashes the data among all the nodes and all communications are conducted in a fully parallel fashion, that is, there is no network congestion. If N is the number of tuples of original data, the communication cost per node is $\frac{N}{k} \cdot \frac{(k-1)}{k}$. The first term of the formula is derived from the fact that each node in a k-node parallel cluster contains $\frac{1}{k}$ of the original data N. When a data-exchange operation executes, $\frac{k-1}{k}$ of the data on each node needs to be transferred to other nodes, and only $\frac{1}{k}$ of the data remains on the same node. Combining these facts yields $\frac{N}{k} \cdot \frac{(k-1)}{k}$. For example, consider a four-node cluster ($k = 4$). Each node contains $\frac{1}{4}$ of the total data N. From each node, $\frac{3}{4}$ of the data on that node needs be transferred to other nodes. Therefore, overall, each of the four nodes transfers $\frac{3}{16}$ of N.

The last column of Table A.1 shows the speed-up of such a communication-bound parallel job. We omit the 1-node cluster case here since there is no communication on one node. We denote the speed-up of the 2-node cluster as 2 as the base in order to have the speed-up metric be normalized based on a number of nodes that starts at 1 (as usual for speed-up). To get the speed-up of the k-node cluster, we first divide the communication cost per node of the 2-node cluster by the communication cost of the k-node cluster. We then multiply the base speed-up of the 2-node cluster, which is 2, by this calculated ratio to get the speed-up of the k-node cluster. For instance, the speed-up of 4-node cluster can be calculated as $\frac{1}{4}/\frac{3}{16} \cdot 2$, which is 2.67.

Figs. A.1(a) and A.1(b) show the comparison between the speed-up of our compu-tation-bound parallel job and ideal linear speed-up. Fig. A.1(a) shows the number of nodes up to 64. We can see there that the trend of the speed-up of a communication-bound parallel job does not saturate. In fact, as we increase the number of nodes, the speed-up becomes $\frac{k}{2}$. In fact, after 12 nodes, the speed-up is always close to $\frac{k}{2}$. When we zoom in to a smaller number of nodes, in Fig. A.1(b), we can see that the speed-up is less than $\frac{k}{2}$, though the speed-up increases gradually as the number of nodes increases.

Appendix B. Parallel sort job

A sort operation takes $O(N \cdot \log N)$ time where N is the amount of data. If there are k nodes in a parallel cluster, a parallel sort job takes $O(\frac{N}{k} \cdot \log \frac{N}{k})$ time on each node since the amount of the data on each node is $\frac{N}{k}$. Thus, we can compute the speed-up of a parallel sort job on the k-node cluster as being $k \cdot \log_{\frac{N}{k}} N$ by simplifying the formula $N \cdot \log N \,/\, \frac{N}{k} \cdot \log \frac{N}{k}$. As shown in Table B.1 and illustrated in Fig. A.1(c), this speed-up is super-linear since the second term ($\log_{\frac{N}{k}} N$) in the formula is always greater than 1 when k is greater than 1. For instance, on a 16-node cluster, the speed-up is 20.02 (>16) when N is 1,000,000.

References

[1] P. Christen, Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection, in: Data-Centric Systems and Applications, 2012.
[2] E. Rahm, H.H. Do, Data cleaning: Problems and current approaches, IEEE Data Eng. Bull. 23 (4) (2000) 3–13.
[3] S.P. Borgatti, A. Mehra, D.J. Brass, et al., Network analysis in the social sciences, Science 323 (5916) (2009) 892–895.
[4] R. Vernica, M.J. Carey, C. Li, Efficient parallel set-similarity joins using MapReduce, in: Proc. 2010 SIGMOD, Indianapolis, IN, USA, 2010, pp. 495–506.
[5] T. Kim, W. Li, A. Behm, et al., Supporting similarity queries in Apache AsterixDB, in: Proc. 21st Int'l Conf. on EDBT, Vienna, Austria, 2018.
[6] C. Li, J. Lu, Y. Lu, Efficient merging and filtering algorithms for approximate string searches, in: Proc. 24th ICDE, Cancun, Mexico, 2008, pp. 257–266.
[7] S. Sarawagi, A. Kirpal, Efficient set joins on similarity predicates, in: Proc. 2004 SIGMOD, Paris, France, 2004, pp. 743–754.
[8] A. Behm, S. Ji, C. Li, J. Lu, Space-constrained gram-based indexing for efficient approximate string search, in: Proc. 25th ICDE, Shanghai, China, 2009, pp. 604–615.
[9] C. Li, B. Wang, X. Yang, VGRAM: Improving performance of approximate queries on string collections using variable-length grams, in: Proc. 33rd Int'l Conf. on VLDB, 2007, pp. 303–314.
[10] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, et al., Approximate string joins in a database (almost) for free, in: Proc. 27th Int'l Conf. on VLDB, San Francisco, CA, USA, 2001, pp. 491–500.
[11] R.J. Bayardo, Y. Ma, R. Srikant, Scaling up all pairs similarity search, in: Proc. 16th Int'l Conf. on WWW, New York, NY, USA, 2007, pp. 131–140, http://dx.doi.org/10.1145/1242572.1242591, URL http://doi.acm.org/10.1145/1242572.1242591.
[12] C. Xiao, W. Wang, X. Lin, et al., Efficient similarity joins for near duplicate detection, in: Proc.of the 17th Int'l Conf. on WWW, Beijing, China, 2008, pp. 131–140.
[13] P. Bouros, S. Ge, N. Mamoulis, Spatio-textual similarity joins, Proc. VLDB Endow. 6 (1) (2012) 1–12.
[14] L.A. Ribeiro, T. Härder, Generalizing prefix filtering to improve set similarity joins, Inf. Syst. 36 (1) (2011) 62–78.
[15] W. Mann, N. Augsten, PEL: position-enhanced length filter for set similarity joins, in: Grundlagen von Datenbanken, in: CEUR Workshop Proceedings, vol. 1313, 2014, pp. 89–94.
[16] C. Xiao, W. Wang, X. Lin, Ed-join: an efficient algorithm for similarity joins with edit distance constraints, Proc. VLDB Endow. 1 (1) (2008) 933–944.
[17] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering?: an adaptive framework for similarity join and search, in: Proc. 2012 SIGMOD, Scottsdale, AZ, USA, 2012, pp. 85–96.
[18] J. Qin, W. Wang, Y. Lu, et al., Efficient exact edit similarity query processing with the asymmetric signature scheme, in: Proc. 2014 ACM SIGMOD, Athens, Greece, 2011, pp. 1033–1044.
[19] W. Wang, J. Qin, C. Xiao, X. Lin, H.T. Shen, Vchunkjoin: An efficient algorithm for edit similarity joins, IEEE Trans. Knowl. Data Eng. 25 (8) (2013) 1916–1929.
[20] D. Deng, G. Li, J. Feng, A pivotal prefix based filtering algorithm for string similarity search, in: Proc. 2014 SIGMOD, 2014, pp. 673–684, Snowbird, Utah, USA.
[21] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: Proc. 23rd Int'l Conf. on VLDB, 1997, pp. 426–435, San Francisco, CA, USA.
[22] J. Feng, J. Wang, G. Li, Trie-join: a trie-based method for efficient string similarity joins, Proc. VLDB Endow. 21 (4) (2012) 437–461.
[23] D. Deng, G. Li, H. Wen, J. Feng, An efficient partition based method for exact set similarity joins, Proc. VLDB Endow. 9 (4) (2015) 360–371, http://dx.doi.org/10.14778/2856318.2856330, http://www.vldb.org/pvldb/vol9/p360-deng.pdf.
[24] Y. Jiang, G. Li, J. Feng, et al., String similarity joins: An experimental evaluation, Proc. VLDB Endow. 7 (8) (2014) 625–636.
[25] W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, Proc. VLDB Endow. 9 (9) (2016) 636–647.
[26] M. Yu, G. Li, D. Deng, et al., String similarity search and join: a survey, Front. Comput. Sci. 10 (3) (2016) 399–417.
[27] Y.N. Silva, J.M. Reed, Exploiting MapReduce-based similarity joins, in: Proc. 2012 SIGMOD, Scottsdale, AZ, USA, 2012, pp. 693–696.
[28] A. Metwally, C. Faloutsos, V-SMART-join: A scalable MapReduce framework for all-pair similarity joins of multisets and vectors, Proc. VLDB Endow. 5 (8) (2012) 704–715.
[29] Y. Wang, A. Metwally, S. Parthasarathy, Scalable all-pairs similarity search in metric spaces, in: Proc. 19th SIGKDD, Chicago, IL, USA, 2013, pp. 829–837.
[30] D. Deng, G. Li, S. Hao, et al., MassJoin: A MapReduce-based method for scalable string similarity joins, in: Proc. 30th ICDE, Chicago, IL, USA, 2014, pp. 340–351.

[31] M.K. Sohrabi, H. Azgomi, Parallel set similarity join on big data based on locality-sensitive hashing, Sci. Comput. Program. 145 (2017) 1–12, http://dx.doi.org/10.1016/j.scico.2017.04.006.
[32] C. Doulkeridis, K. Nørvåg, A survey of large-scale analytical query processing in MapReduce, Proc. VLDB Endow. 23 (3) (2014) 355–380.
[33] L. Gravano, P.G. Ipeirotis, N. Koudas, et al., Text joins in an RDBMS for web data integration, in: Proc. 12th Int'l Conf. on WWW, Budapest, Hungary, 2003, pp. 90–101, http://dx.doi.org/10.1145/775152.775166, URL http://doi.acm.org/10.1145/775152.775166.
[34] S. Chaudhuri, V. Ganti, R. Kaushik, Data debugger: An operator-centric approach for data quality solutions, IEEE Data Eng. Bull. 29 (2) (2006) 60–66.
[35] Y.N. Silva, W.G. Aref, M.H. Ali, The similarity join database operator, in: Proc. 26th ICDE, Long Beach, CA, USA, 2010, pp. 892–903.
[36] Y.N. Silva, S.S. Pearson, J. Chon, R. Roberts, Similarity joins: Their implementation and interactions with other database operators, Inf. Syst. 52 (2015) 149–162.
[37] J. Sun, Z. Shang, G. Li, et al., Dima: A distributed in-memory similarity-based query processing system, Proc. VLDB Endow. 10 (12) (2017) 1925–1928.
[38] P. Jokinen, E. Ukkonen, Two algorithms for approxmate string matching in static texts, in: Proc. 16th Int'l Symp. on MFCS, Kazimierz Dolny, Poland, 1991, pp. 240–248.
[39] S. Alsubaiee, Y. Altowim, H. Altwaijry, et al., AsterixDB: A scalable, open source BDMS, Proc. VLDB Endow. 7 (14) (2014) 1905–1916, URL http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf.
[40] APACHE AsterixDB, http://asterixdb.apache.org.
[41] K.W. Ong, Y. Papakonstantinou, R. Vernoux, The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases, CoRR abs/1405.3631 (2014).
[42] V.R. Borkar, Y. Bu, E.P.C. Jr., et al., Algebricks: A data model-agnostic compiler backend for big data languages, in: Proc. SCC, Kohala, HI, USA, 2015, pp. 422–433.
[43] V.R. Borkar, M.J. Carey, R. Grover, et al., Hyracks: A flexible and extensible foundation for data-intensive computing, in: Proc. 27th ICDE, 2011, pp. 1151–1162.
[44] S. Alsubaiee, A. Behm, V.R. Borkar, et al., Storage management in AsterixDB, Proc. VLDB Endow. 7 (10) (2014) 841–852, http://dx.doi.org/10.14778/2732951.2732958.
[45] D. Kossmann, The state of the art in distributed query processing, ACM Comput. Surv. 32 (4) (2000) 422–469.
[46] J. McAuley, R. Pandey, J. Leskovec, Inferring networks of substitutable and complementary products, in: Proc. 21st Int'l Conf. on KDD, Sydney, Australia 2015, pp. 785–794.
[47] Reddit Dataset, https://reddit.com/r/datasets/comments/3mg812/.
[48] Twitter Streaming API, https://developer.twitter.com.
[49] Source code - Apache Hadoop Map/Reduce version of the three-stage-similarity join, http://asterix.ics.uci.edu/fuzzyjoin/.
[50] Elastic Search, https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html.
[51] C. Gormley, Z. Tong, Elasticsearch: The Definitive Guide, first ed., O'Reilly Media, Inc., 2015.
[52] S. Ji, G. Li, C. Li, J. Feng, Efficient interactive fuzzy keyword search, in: WWW, 2009, pp. 371–380.