Category- and Selection-Enabled Nearest Neighbor Joins

Francesco Cafagna*,[1,2], Michael H. Böhlen[1]

[1] *Department of Computer Science, University of Zürich, Switzerland*

[2] *AdNovum Informatik, Switzerland*

Annelies Bracher

Agroscope, Switzerland

Abstract

This paper proposes a *category- and selection-enabled* nearest neighbor join (NNJ) between relation **r** and relation **s**, with similarity on T and support for category attributes **C** and selection predicate θ. Our solution does not suffer from *redundant fetches* and *index false hits*, which are the main performance bottlenecks of current nearest neighbor join techniques.

A *category-enabled* NNJ leverages the category attributes **C** for query evaluation. For example, the categories of relation **r** can be used to limit relation **s** accessed at most once. Solutions that are not category-enabled must process each category independently and end up fetching, either from disk or memory, the blocks of the input relations multiple times. A *selection-enabled* NNJ performs well independent of whether the DBMS optimizer pushes the selection down or evaluates it on the fly. In contrast, index-based solutions suffer from many index false hits or end up in an expensive nested loop.

Our solution does not constrain the physical design, and is efficient for row- as well as column-stores. Current solutions for column-stores use late materialization, which is only efficient if the data is clustered on the category attributes **C**. Our evaluation algorithm finds, for each outer tuple r, the inner tuples that satisfy the equality on the category and have the smallest distance to r with only one scan of both inputs. We experimentally evaluate our solution using a data warehouse that manages analyses of animal feeds.

Keywords: Robust Nearest Neighbor Join; Similarity Join; Sort Merge; Query Optimization; Column-store; PostgreSQL

*Corresponding author

Email addresses: `cafagna@ifi.uzh.ch` (Francesco Cafagna*,[1,2]), `boehlen@ifi.uzh.ch` (Michael H. Böhlen[1]), `annelies.bracher@agroscope.admin.ch` (Annelies Bracher)

1. Introduction

In most real world applications with nearest neighbor joins (NNJs), the nearest neighbors of a tuple $r \in \mathbf{r}$ must be determined for a subset of the tuples in relation $\mathbf{s}$. As an example, consider a data warehouse with a fact table $\mathbf{s}$ that stores analyses of animal feeds. If an application asks for the 'Vitamin A' value of 'Soy' on 2014-05-01, we must find the analyses in $\mathbf{s}$ with the timestamp closest to 2014-05-01, but only among the tuples that *i*) satisfy *predicate* Nutrient = 'Vitamin A', and *ii)* have the same *category* value (i.e., we want a value for 'Soy'). Towards this end, we propose a category- and selection-enabled NNJ operator, which, for each tuple in relation $\mathbf{r}$, returns the tuple(s) in $\mathbf{s}$ with the most similar value for T that have the same category $\mathbf{C}$ and satisfy predicate θ.

In the past, efficient solutions have been developed for computing NNJs. These solutions are neither *category-* nor *selection-enabled* and become inefficient if $\mathbf{C}$ or θ is present. This is a non-trivial problem in NNJ queries since the equality on $\mathbf{C}$ and the evaluation of θ cannot be postponed until after the NNJ [1]. For example, a NNJ might select as nearest neighbor of ('Soy', 2014-05-01) the tuple ('Pea', 2014-05-02). Clearly, this pair is filtered out after evaluating the equality on $\mathbf{C}$ since the categories are different. Thus, no nearest neighbor for ('Soy', 2014-05-01) would be returned, which is incorrect.

Our approach is the first NNJ solution that is *category- and selection-enabled.* It efficiently deals with selective predicates and multiple categories, without constraining the physical organization of the input relations. Our *category-enabled* algorithm is run once independent of the number of categories. As a result, our query tree copes well with any optimization on the category attributes to reduce the runtime. For example, in a data warehouse context, when the DBMS uses the categories in the query to limit the tuples of the fact table, our approach still fetches each block of the fact table at most once. This is not the case when a SortMerge NNJ [2], in combination with the SegmentApply operator [3] to handle different categories, is used. For such an approach, if a disk or memory block stores tuples of different categories, it is fetched multiple times. Our *selection-enabled* approach stays robust if either the DBMS optimizer pushes down the evaluation of the predicate below the NNJ (e.g., if θ is selective), or if it evaluates the selection on the fly (e.g., in case θ is often true). Current B-Tree solutions [4] deteriorate if the selection is evaluated on the fly since, if the closest tuple fetched does not satisfy the selection predicate, then the second closest must be retrieved, then the third closest, and so on until a tuple satisfying the predicate is fetched.

Our approach does not suffer if a given block stores tuples of different categories since the blocks of the input relations are accessed at most once, independent of the number of categories that are stored on a block. Our solution also does not suffer if θ is selective since, in such a case, the tuples that do not satisfy θ are filtered out before the NNJ. The robustness of our solution is independent of the physical design. For example, column-stores perform well only if a primary (or clustered) index on the relations is available: if the data is not clustered by ($\mathbf{C}$, T), redundant fetches are computed on $\mathbf{C}$, T, and on every column involved in θ. Our approach does not require any clustering or index structure, but indexes are leveraged to directly access the tuples. The independence of the physical design is a key property of category- and selection-enabled NNJs for two reasons: first, only one clustering can exist; and second, the category and similarity attributes are query dependent and change for each query (one NNJ query might compute the similarity on the price, another one on the time, and yet another on the quantity). In our experiments, we show that our approach is up to two orders of magnitude faster than state of the art solutions for computing real world queries on the Swiss Feed Data Warehouse [5] and on the GREEND dataset [28] if no primary index for the category and similarity attributes is available.

Our technical contributions are as follows:

- We introduce and define the category- and selection-enabled NNJ operator.
- We introduce an efficient query tree to compute queries with category- and selection-enabled NNJs. Our query tree can be integrated both in row- and column-stores. Independent of the clustering of the input relations, our solution does not suffer from redundant fetches and false hits.
- We provide *roNNJ*, a sort-merge-based algorithm that, for each tuple of the left subtree, finds the tuples in the right subtree that have the same category and the closest values of the similarity attribute, with a single scan of both inputs.
- We describe the seamless integration of NNJ queries with predicates and categories into PostgreSQL.
- We use the Swiss Feed Data Warehouse and the GREEND dataset to experimentally evaluate the performance of our approach and compare it with the state of the art techniques implemented on disk, main memory, and column-stores.

This paper extends the work in Cafagna et al. [9]. Beyond the contributions of this work we explore the advantages of a query tree with our category-enabled NNJ, and we show that the drawbacks of related approaches are independent of the physical design of the input relations. Towards this goal we implemented and evaluated our solution on column-stores. We show how category- and selection-enabled NNJs can be integrated into the query trees of a column-store (e.g., MonetDB). The experimental evaluation compares the runtime on column-stores against ours. Due to an increase of the size of our dataset (new analyses have been added to the Swiss Feed Data Warehouse during the last year), the absolute numbers in the experiments differ from the ones in [9], especially for the B-Tree since the number of look-ups to compute (two per outer tuple) has increased.

The paper is organized as follows. In Section 2 we present our running example. Section 3 discusses related work. Section 4 defines the category-

and selection-enabled NNJ. In Section 5 we describe our algorithm. Section 6 introduces NNJ query trees and offers an analytical evaluation of our approach. Section 7 reports the result of an empirical evaluation on the Swiss Feed Data Warehouse and the GREEND dataset. Section 8 draws conclusions and points to future work.

2. Running Example

As a running example, we use the Swiss Feed Data Warehouse [5], i.e., a data warehouse that stores lab analysis of animal feeds, using a fact table with a vertical design where each value of the analysis is stored in a different row [7], [8]. Figure 1 shows selected tuples of fact table **s**. Animal feeds C, such as 'Soy', 'Pea', or 'Hay', are sampled in the field at an altitude A and analyzed at time T in a lab where the value V of various nutrients N is measured with reliability R. For instance, tuple s_0 records that for feed 'Soy', grown at an altitude of 1030 meters, the nutrient content of 'CP' (Crude Protein) at time 2014-06-15 is 1.40 with reliability 0.9. Since lab analyses are expensive and more than 600 nutrients exist, not all nutrients N are measured on a daily basis (e.g., no 'CP' value has been measured for 'Soy' on 2014-06-19).

r

	C	T
r_0	Soy	2014-06-15
r_1	Soy	2014-06-21
r_2	Pea	2014-06-20

s

	C	T	A	R	N	V
s_0	Soy	2014-06-15	1030	0.9	CP	1.40
s_1	Soy	2014-06-20	1000	1.0	CP	1.08
s_2	Soy	2014-06-21	1020	0.5	CP	0.93
s_3	Soy	2014-06-27	1110	0.9	CP	1.23
s_4	Pea	2014-06-19	1000	0.8	CP	4.20
s_5	Pea	2014-06-20	1000	0.3	CP	4.10
s_6	Pea	2014-06-21	1100	0.9	CP	4.03
s_7	Hay	2014-06-19	1000	0.8	OM	0.32

Figure 1: Outer Relation **r**; Fact Table **s** with Lab Analyses of the Nutrients of Feeds.

Relation **r** in Figure 1 illustrates the outer tuples for which the nearest neighbors in **s** must be retrieved: attributes C and T correspond to the feed and day for which a measurement is needed. In the web application of the Swiss Feed Data Warehouse (`http://www.feedbase.ch`), the users (farmers, domain experts, etc.) use the result of the NNJs to compute graphical interpolations that represent the evolution of a given nutrient in different feeds to pick the feed that best suits the desired characteristics (e.g., the *cereal* that has the most stable protein content in the animal feeding process). Attribute C typically covers up to 5% of the feeds stored in the fact table (e.g., all cereals), while attribute T represents the days for which the nutritive values must be computed. Note that it is not possible to precompute the join off-line since the result depends on predicate θ, which is defined over a combination of attributes that change for each query, e.g., $\theta \equiv (N = \text{`CP'} \wedge R > 0.7)$.

Table 1 summarizes the notation we use in this paper.

Table 1: Notation.

Symbol	Meaning	Example
r	tuple	r, r_i, s, s_j, z_1
$\mathbf{r}$	relation	$\mathbf{r}, \mathbf{s}, \mathbf{z}$
$\|\mathbf{r}\|$	cardinality	$\|\mathbf{r}\|, \|\mathbf{s}\|, \|\mathbf{z}\|$
$\mathbf{C}$	set of category attributes	C
T	similarity attribute	T
θ	predicate	$R > 0.7$
$sel(\theta)$	predicate selectivity	$\frac{\|\sigma_{R>0.7}(\mathbf{s})\|}{\|\mathbf{s}\|}$
$sel(\mathbf{C})$	category selectivity	$\frac{\|\sigma_{\mathbf{C}\in\pi_{\mathbf{C}}(\mathbf{r})}(\mathbf{s})\|}{\|\mathbf{s}\|}$
$\mathbf{r} \ltimes^T \mathbf{s}$	NNJ	$\mathbf{r} \ltimes^T \mathbf{s}$
$\mathbf{r} \ltimes^T[\mathbf{C}, \theta]\ \mathbf{s}$	category- and selection-enabled NNJ	$\mathbf{r} \ltimes^T[C, R > 0.7]\ \mathbf{s}$

3. Related Work

In this section, we introduce the state of the art NNJ solutions and explain the problems they face when dealing with categories and predicates.

3.1. B-Tree

Yao et al. [4] proposed an implementation for $r \ltimes^T \mathbf{s}$, i.e., a NNJ without categories and selections, using a B-tree. The solution can easily leverage categories and compute a NNJ with a B-tree on $(\mathbf{s}.\mathbf{C}, \mathbf{s}.T)$. This approach performs, for each $r \in \mathbf{r}$, two index look-ups in $\mathbf{s}$ using $(r.\mathbf{C}, r.T)$ as search key (Figure 2.a). One lookup fetches the first tuple to the left (using a MAX subquery), and one lookup fetches the first tuple to the right (using a MIN subqery). The closer of the two tuples is the nearest neighbor of r. The approach is elegant since it does not require any change in the DBMS engine and can be implemented using SQL statements. As for any indexed method, this approach performs particularly well if $|\mathbf{r}| \ll |\mathbf{s}|$. However, as shown in Figure 2(b), this approach

Figure 2: B-Tree implementation where, for $r \in \mathbf{r}$, the nearest neighbor is found with a MAX and a MIN query using a B-Tree on $(\mathbf{s}.\mathbf{C}, \mathbf{s}.T)$. The second figure highlights in grey the false hits of the tuples that do not satisfy predicate θ.

suffers from index false hits if a predicate θ is present. For example the MAX subquery:

```
SELECT MAX(T) AS sMax
FROM s
WHERE s.C = r.C AND s.T < r.T AND θ
```

no longer guarantees that the maximum is the first tuple that is reached through the index. The index must be scanned to the left until a tuple satisfying θ is found. The higher the selectivity of θ, the higher the number of false hits.

This drawback is even more pronounced in column-stores. Although column-stores do not fetch an entire tuple that does not satisfy θ, a single false hit affects each column involved in θ, i.e., for each column a different block must be fetched to evaluate the corresponding predicate. This means that in a column-store each index false hit results in multiple block fetches (and not just in one as for row-stores). We show in our experiments that, while Apache Cassandra [10] adopts this plan if a primary index on $(\mathbf{C}, T)$ exists[1], MonetDB [11] chooses a different plan: it first fetches the OIDs of the tuples with the same category as r; then T (and each column involved in θ) is scanned and only the OIDs of the entries satisfying $\mathbf{s}.T < r.T$ (or predicate θ) are kept; afterwards, the intersection of the OIDs returned from the previous selections is computed; finally, the MAX on the returned tuples is computed. We show in our experiments that this plan, although it avoids the index false hits, is similar to a nested-loop since for a given $r \in \mathbf{r}$ it fetches from $\mathbf{s}$ all the entries with the same category, and it is therefore expensive.

3.2. SegmentApply

Silva et al. [2] proposed a NNJ operator that is not category-enabled, i.e., it computes $\mathbf{r} \ltimes^{T} \mathbf{s}$, using SortMerge. This approach sorts $\mathbf{r}$ and $\mathbf{s}$ by T, and computes the merge step with a single scan of the relations by taking advantage of the order of the tuples. Opposite to indexed methods, this approach scales to large datasets. However, as shown in [1], the equality on the categories cannot just be evaluated after a category-unaware NNJ[2]. To manage multiple categories, the SegmentApply operator [3] must be used. It is implemented in DBMSs as lateral subqueries that fetch, for each category $c \in \pi_{\mathbf{C}}(\mathbf{r})$, the input tuples with category c and run a SortMerge NNJ:

```
SELECT *
FROM (SELECT DISTINCT $\mathbf{C}$ FROM $\mathbf{r}$) $\mathbf{c}$,
     LATERAL (SELECT * FROM $\mathbf{r}$ WHERE $\mathbf{C} = \mathbf{c}.\mathbf{C}$) $\mathbf{r}$
     NNJ
     LATERAL (SELECT * FROM $\mathbf{s}$ WHERE $\mathbf{C} = \mathbf{c}.\mathbf{C}$ AND $\theta$) $\mathbf{s}$
     ON T
```

This approach suffers from redundant fetches since it requires a scan of the input relations for each category in $\mathbf{r}$. Note that also in the presence of an index on $\mathbf{s}.\mathbf{C}$ redundant fetches happen since, if a block stores tuples of m required categories, this block is fetched redundantly m times, once for each category.

[1]In Cassandra the previous query needs to be rewritten as `SELECT` $\mathbf{C}, T$ `FROM` $\mathbf{s}$ `WHERE` $\mathbf{C} = r.\mathbf{C}$ `AND` $T \leq r.T$ `AND` θ `ORDER BY` T `LIMIT 1 ALLOW FILTERING` in order to take advantage of the primary index.

[2]The computation of `SELECT * FROM r NNJ s ON T WHERE` $\mathbf{r}.\mathbf{C} = \mathbf{s}.\mathbf{C}$ is not correct, since it first joins each $r \in \mathbf{r}$ with its nearest neighbor in $\mathbf{s}$ (independently on its category), and then filters out the joined tuples with different categories.

In column-stores, for each category $c \in \pi_{\mathbf{C}}(\mathbf{r})$, $\mathbf{s}.\mathbf{C}$ is accessed and the OIDs of the tuples of category c are returned. The OIDs are joined with the (OID, Value) pairs of each column involved in θ, to select only the tuples that satisfy θ. Finally, their T value is fetched and a category-unaware sort-merge NNJ is run. Without a primary index, i.e., when the input relations are not clustered on $\mathbf{C}$, redundant fetches occur since, if m (OID, Value) pairs are stored in the same block and refer to tuples of different categories, then this block is fetched m times. The remaining columns are sequentially scanned and joined to the result after the NNJ for every category is computed, and incur no redundant fetches. If a primary index on $\mathbf{s}.\mathbf{C}$ is present, the redundant fetches are almost eliminated, since the (OID, Value) pairs of each column are clustered based on category $\mathbf{C}$. This means that, for each column, the entries of the same category are placed in contiguous blocks. While processing the i-th category, only the first processed block of each column (storing also tuples of the $(i-1)$-th category) will be read redundantly. Note that the category attributes $\mathbf{C}$ are query-dependent: for example, in the Swiss Feed Database, depending on the query, the category attributes might specify a biological column (feed type, feed name, stage of maturity, etc.), a geographical column (country, region, postal code, etc.), etc. We include both scenarios in our experiments, and show that our category- and selection-enabled NNJ is the only technique that is robust in cases where no clustered index for the category attribute exists.

3.3. Category-Based Optimization Rules

Kimura et al. [12] introduced the SortedIndexScan to efficiently compute indexed selections on the categories of the query points, i.e., $\sigma_{\mathbf{C} \in \pi_{\mathbf{C}}(\mathbf{r})}(\mathbf{s})$. This technique traverses the index on $\mathbf{s}.\mathbf{C}$ for each needed category (i.e., $\pi_{\mathbf{C}}(\mathbf{r})$) and keeps a list of the block IDs that store matching tuples. The block IDs are sorted and deduplicated to avoid fetching multiple times the same block. An equivalent technique has been introduced in the Orca query optimizer [13]. The Orca query optimizer reduces the number of partitions to fetch, i.e., the relevant blocks, in multi-level partitioned fact tables [14]: for each dimension table involved in the join, a *PartitionSelector* scans it and keeps a list of IDs of the partitions of $\mathbf{s}$ with join matches. At the end, the intersection of the lists is passed to the *DynamicScanner*, which reads the relevant blocks. We show that our category-enabled query tree can fully leverage such optimizations. As a result, it fetches only the blocks that store tuples with relevant categories (i.e., the categories of $\mathbf{r}$), and it does so once.

3.4. Similarity Joins

During the last few years, different similarity join operators have been proposed [15]: *k-nearest neighbor joins* (k-NNJ) where each outer tuple is joined with the k closest inner tuples, *ϵ joins* where each outer tuple is joined with all tuples within a given distance range ϵ [16], *k-distance joins* where the k closest pairs are retrieved, *join around* where the result is the intersection of an ϵ join and a k-nearest neighbor join, and a reverse k-nearest neighbor join [17] which determines the tuples that have the query points as one of their k-nearest neighbors. None of those operators integrates categories and predicates.

Works on the evaluation of query trees combining two k-NNJ queries, have been studied by Aly et al. [18]. Note that our implementation assumes $k = 1$ for simplicity but, if many tuples are found at the same minimum distance, returns all of them. Our implementation can be easily adapted to a k-NNJ by using a fixed-size window (of length k) of nearest neighbors.

Partition-based solutions for computing similarity joins have been proposed in the context of Quickjoin and D-Index. Quickjoin [19] partitions the data space according to pivot points and creates windows to bind adjacent partitions. Partitions are recursively sub-partitioned until they are small enough to be processed in a memory nested loop. D-Index [20] partitions according to a set of mapping functions ρ and uses the D-Index and its extension called eD-Index [21] to access a small portion of data within which the closest pairs are searched. Quickjoin focuses on ϵ joins and requires a rebuild of the index for each different ϵ value; D-Index has been introduced for computing self-joins. Both approaches do not consider categories and predicates, and have not been integrated into a DBMS.

4. A Category- and Selection-Enabled Nearest Neighbor Join

We assume a multidimensional schema $\boldsymbol{S} = [\mathbf{C}, T, \mathbf{V}]$ with attributes $C_1, ..., C_m, T, V_1, ..., V_n$. We write $\mathbf{s}$ to indicate a relation over schema $\boldsymbol{S}$, and $|\mathbf{s}|$ for the number of tuples in $\mathbf{s}$. For a tuple $s \in \mathbf{s}$ and an attribute V_i, $s.V_i$ denotes the value of attribute V_i. We assume a totally ordered similarity attribute T. The concatenation operator $r \circ s$ appends to r the attributes of s.

Definition 1. Assume relation $\mathbf{r}$ with schema $\boldsymbol{R} = [\mathbf{C}, T]$ and relation $\mathbf{s}$ with schema $\boldsymbol{S} = [\mathbf{C}, T, \mathbf{V}]$. Let θ be a predicate on $\boldsymbol{S}$. The *Category- and Selection-Enabled Nearest Neighbor Join*, $\mathbf{r} \ltimes^{T}[\mathbf{C}, \theta]\, \mathbf{s}$, returns, for a given tuple $r \in \mathbf{r}$, the tuples $s \in \sigma_\theta(\mathbf{s})$ with the same category $\mathbf{C}$ that have the closest T value:

$$\mathbf{r} \ltimes^{T}[\mathbf{C}, \theta]\;\; \mathbf{s} = \big\{ r.\boldsymbol{R} \circ s.\mathbf{V} \mid r \in \mathbf{r} \wedge s \in \sigma_\theta(\mathbf{s}) \;\wedge\; r.\mathbf{C} = s.\mathbf{C} \;\wedge \\ \nexists\, t \in \sigma_\theta(\mathbf{s}) \big(r.\mathbf{C} = t.\mathbf{C} \wedge |r.T - t.T| < |r.T - s.T| \big) \big\}$$

Example 1. Consider in Figure 1 relations $\mathbf{r}$ with schema $\boldsymbol{R} = [C, T]$ and $\mathbf{s}$ with schema $\boldsymbol{S} = [C, T, A, R, N, V]$, and predicate $\theta \equiv (N = \text{'CP'} \wedge R > 0.7)$. We apply Definition 1 with $\mathbf{C} = C$ and $\mathbf{V} = A, R, N, V$ to compute $\mathbf{z} = \mathbf{r} \ltimes^{T}[C, N = \text{'CP'} \wedge R > 0.7]\, \mathbf{s}$. Thus, we join each outer tuple in $\mathbf{r}$ with the temporally closest 'CP' measure having reliability greater than 0.7. Result relation $\mathbf{z}$ is shown in Figure 3. For example, the nearest neighbor of tuple r_0 is tuple s_0 since it is the closest. For tuple r_1, s_1 is its nearest neighbor. Even if temporally closer, s_2 has not been chosen as nearest neighbor since it does not satisfy $R > 0.7$. For tuple r_2, two tuples (s_4, s_6) that satisfy $R > 0.7$ exist at the same minimum distance, and therefore two join matches are returned.

$\mathbf{z} = \mathbf{r} \ltimes^{T} [C, N = \text{‘CP’} \wedge R > 0.7]\, \mathbf{s}$

	C	T	A	R	N	V
$r_0 s_0$	Soy	2014-06-15	1030	0.9	CP	1.40
$r_1 s_1$	Soy	2014-06-21	1000	1.0	CP	1.08
$r_2 s_4$	Pea	2014-06-20	1000	0.8	CP	4.20
$r_2 s_6$	Pea	2014-06-20	1100	0.9	CP	4.03

Figure 3: NNJ Result **z**.

The NNJ in our example uses the feed name as a category attribute and the time as similarity attribute. For the similarity attribute T any attribute whose domain is totally ordered can be used (e.g., similarity on the price, the quantity, the time, etc.)

5. The Robust NNJ Algorithm

This section describes the *roNNJ* algorithm, i.e., the implementation of our category- and selection-enabled NNJ into the kernel of PostgreSQL. We give the details of the extension, and present an efficient algorithm that: 1) computes the join with a single access of the input relations, i.e., without redundant fetches; and 2) does not suffer from false hits, i.e., each tuple that is not a nearest neighbor is not read more than once.

5.1. Algorithm Properties

Sort merge has been proposed as a method for computing equijoins [26] and as a method for computing nearest neighbor joins [2]. We describe an implementation that efficiently combines these two approaches (equijoin on the category, and nearest neighbor join on the similarity attribute), and that does not do unnecessary backtracking. As reference point for the input relations we use the tree in Figure 6(b), which applies selection push-down to **s** as a DBMS optimization on **C** and θ.

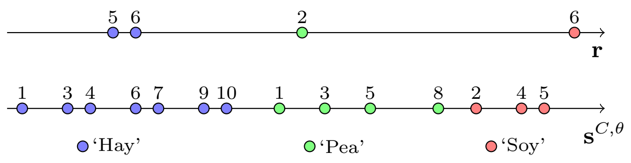


Figure 4: Relations **r** and $\mathbf{s}^{C,N=\text{'CP'}} \equiv \sigma_{C \in \pi_C(\mathbf{r}) \wedge N=\text{'CP'}}(\mathbf{s})$ are sorted by category C (represented by the colour) and similarity attribute T (represented by the number). No tuple may be available for a given timestamp: for example, a gap between (‘Pea’,1) and (‘Pea’, 3) shows that no ‘Pea’ tuple is available for $T = 2$.

The algorithm takes the two input relations $\mathbf{r}$ and $\mathbf{s}^{\mathbf{C},\theta} \equiv \sigma_{\mathbf{C} \in \pi_{\mathbf{C}}(\mathbf{r}) \wedge \theta}(\mathbf{s})$, and sorts them by $(\mathbf{C}, T)$. The sorting by $\mathbf{C}$ guarantees that the tuples are grouped according to their category value. Thus, as illustrated in Figure 4, all tuples of feed 'Hay' (as well as all 'Pea' and 'Soy') are adjacent. This means that, while processing an outer tuple $r \in \mathbf{r}$ with category 'Pea', no backtracking in $\mathbf{s}$ to the tuples of category 'Hay' is needed since the nearest neighbors must satisfy the equality on the categories. The sorting by T makes sure that tuples in $\mathbf{s}^{\mathbf{C},\theta}$ that have been read previously but that were not nearest neighbors do not have to be read again. For example, for $r = (\text{'Hay'}, 5)$, the inner relation $\mathbf{s}^{\mathbf{C},\theta}$ is scanned until its nearest neighbors $(\text{'Hay'}, 4)$ and $(\text{'Hay'}, 6)$ are reached. For the following $\mathbf{r}$ tuple, no tuple before $(\text{'Hay'}, 4)$ has to be considered from $\mathbf{s}^{\mathbf{C},\theta}$ since it will have a higher distance than $(\text{'Hay'}, 4)$ itself. We now prove that our algorithm does not fetch more than once any tuple that is not nearest neighbor.

Lemma 1. *(No Unnecessary Backtracking) The computation of* $\mathbf{r} \ltimes^{T} [\mathbf{C}, \theta]\, \mathbf{s}^{\mathbf{C},\theta}$ *reads each tuple that is not a nearest neighbor at most once.*

Proof. In the merge phase of a sort-merge computation, tuples might be read more than once only during *backtracking* (i.e., going back to a previously scanned row). We now show that backtracking, when applied, fetches only the nearest neighbors. Let $\{r_{i-1}, r_i\} \subseteq \mathbf{r}$, r_i be the current tuple for which the nearest neighbors have to be found, and $d(r_i, s) = |r_i.T - s.T|$ be the absolute distance between tuples r_i and s. Since the input relations are sorted by $\mathbf{C}, T$ three cases are possible:

1. $r_{i-1}.\mathbf{C} < r_i.\mathbf{C}$: straightforward. Since the nearest neighbors of r_i must have its same category, then they will, for sure, be in $\mathbf{s}$ after all tuples of category $r_{i-1}.\mathbf{C}$. No backtracking needs to be applied.
2. $r_{i-1}.\mathbf{C} = r_i.\mathbf{C} \wedge r_{i-1}.T < r_i.T$. Let s_j be the first nearest neighbor of r_{i-1}: since r_i succeeds r_{i-1}, then $d(r_i, s_j) = d(r_{i-1}, s_j) + d(r_{i-1}, r_i)$. For any $k < j$ such that s_k has the same category as r_i (if s_k has a different category, case 1 re-applies), $r_i.\mathbf{C} = s_k.\mathbf{C} \Rightarrow d(r_i, s_k) = d(s_k, s_j) + d(s_j, r_{i-1}) + d(r_{i-1}, r_i)$. Since $s_k.T < s_j.T$, we get $d(s_k, s_j) > 0$: any tuple s_k preceding the first nearest neighbor (s_j) of r_{i-1} will have a bigger distance to r_i than s_j itself, i.e., $d(r_i, s_k) > d(r_i, s_j)$. No backtracking needs to be applied.
3. $r_{i-1}.\mathbf{C} = r_i.\mathbf{C} \wedge r_{i-1}.T = r_i.T$. Tuples r_{i-1} and r_i share the same nearest neighbors, and backtracking to the nearest neighbors of r_{i-1} *has* to be applied. Let s_j and s_k be, respectively, the first and the last nearest neighbors of r_{i-1}. In order for Lemma 1 to hold, we must make sure that between s_j and s_k no tuple exists that is not a nearest neighbor for r_i. Since s_j and s_k are nearest neighbors, they must have the same (minimum) distance from r_i, i.e., $d(r_i, s_j) = d(r_i, s_k)$. However, this is true only if $d(r_i, s_j) = d(r_i, s_k) \Leftrightarrow s_j.T = (r_i.T - \epsilon) \wedge s_k.T = (r_i.T + \epsilon)$, with $\epsilon \geq 0$. A tuple $s \in \mathbf{s}$ such that $r_i.T - \epsilon < s.T < r_i.T + \epsilon$ cannot exists, otherwise it would be nearest neighbor itself, instead of s_j and s_k.

The above Lemma proves that our approach computes a NNJ with only one scan of the input relations. The only tuples that our algorithm rescans through backtracking are the nearest neighbors (if two outer tuples r_i and r_{i+1} have the same result tuples). For example, for tuple r_2 of our main example, s_4 and s_6 are the nearest neighbors. Removing s_5 before the sorting, ensures that s_4 and s_6 will be adjacent elements in $\mathbf{s}$. Backtracking would only be needed if a tuple r_3 existed with s_4 and s_6 as its nearest neighbors.

5.2. The roNNJ Algorithm

We now describe the *roNNJ* implementation of the NNJ operator. The SortMerge *roNNJ* has been implemented as a set of states (cf. Figure 5), similar to the traditional sort merge joins in commercial DBMSs.

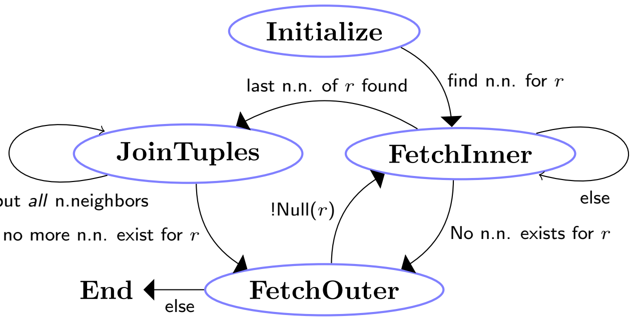


Figure 5: State diagram of Exec_roNNJ

If the actual node to compute is a NNJ, the executor of the DBMS calls a procedure **Exec_roNNJ**(*NNJObj*), where *NNJObj* is an object shared by all states, storing, among others: ***OuterPlan** (a reference to the outer tuples), ***InnerPlan** (a reference to the inner tuples), **C** (the position of the category attributes in the schema of **r** and **s**), T (the position of the similarity attribute), r (the current outer tuple for which the nearest neighbors have to be found), s_c (the current inner tuple), s_n (the next inner tuple), *nextState* (the state to be executed in the next iteration of the algorithm). Thus, opposite to an equijoin node where only the current tuples are stored, for a NNJ both the current and the next **s** tuples are needed. This is so since, after the sorting, we can decide if s_c is the nearest neighbor of r only after comparing its distance with the one of s_n.

The procedure is shown in Algorithm 1: it consists of a loop in which, at each iteration, one state is executed and the object *NNJObj* is modified. *NNJObj* is initially set to $(\mathbf{r}, \mathbf{s}^{\mathbf{C},\theta}, \mathbf{C}, T, null, null, null, 1)$, where $\mathbf{s}^{\mathbf{C},\theta} \equiv \sigma_{\mathbf{C} \in \pi_{\mathbf{C}}(\mathbf{r}) \wedge \theta}(\mathbf{s})$, i.e., the right subtree of Figure 6. In each state tuples are fetched, joined, etc., and the next state to be executed is set. For conciseness, we omit the name of the object *NNJObj* in front of each variable, e.g., we write r instead of *NNJObj.r*.

Algorithm 1: Exec_roNNJ

```
Called as: Exec_roNNJ($\mathbf{r}, \mathbf{s}^{\mathbf{C},\theta}, \mathbf{C}, T, null, null, null, 1$)
Input    : NNJObj.{*OuterPlan,          // r sorted by $\mathbf{C}, T$
                   *InnerPlan,          // s sorted by $\mathbf{C}, T$
                   $\mathbf{C}$,                 // category attribute
                   $T$,                 // similarity attribute
                   $r$,                    // current r tuple
                   $s_c$,                    // current s tuple
                   $s_n$,                       // next s tuple
                   nextState}          // state to perform next
1 begin
2     while $nextState \leq 4$ do
3         switch nextState do
4             case 1: Initialize(NNJObj)
5             case 2: FetchInner(NNJObj)
6             case 3: JoinTuples(NNJObj)
7             case 4: FetchOuter(NNJObj)
```

The following subsections describe the four states of *roNNJ*. For a given $r \in \mathbf{r}$, *roNNJ* returns *all* its nearest neighbors, i.e., all tuples minimizing their distance to r, independent of their number.[3]

5.2.1. Initialize

In this state, we start the scan of the two relations. We initialize r with the first outer tuple, and s_c and s_n with the first inner tuple. The next state to be computed is FetchInner.

State 1: Initialize(*NNJObj*)

```
1 $r \leftarrow$ fetchRow(OuterPlan)
2 $s_c \leftarrow$ fetchRow(InnerPlan)
3 markPosition(s)
4 $s_n \leftarrow s_c$
5 $nextState = 2$                              // Go to FetchInner
```

5.2.2. FetchInner

In this state we fetch the next inner tuple. First, in lines 1-2 we check if an inner tuple with the same category as the actual outer tuple exists at all: if not, no join match exists for r, and we go to FetchOuter. In lines 4-8 we fetch a new

[3]For k-NNJ queries, joining r with the k closest tuples, a similar implementation with a window of k $\mathbf{s}$ tuples from s_c to s_n can be used. The window is moved until s_n is more far than s_c to r or has a different category.

inner tuple: if r is closer to s_n than to s_c, then s_n *might* be its (first) nearest neighbor. Therefore we mark its position. In lines 9-12, as soon as the next tuple s_n has a higher distance to r or belongs to a different category, we are sure that the previously marked tuple is its first nearest neighbor and no more nearest neighbors exist. We, therefore, restore s_c to the first nearest neighbor and we go to state JoinTuples.

State 2: FetchInner(*NNJObj*)

```
1  if $r.\mathbf{C} < s_c.\mathbf{C}$ then
2      $nextState = 4$                                   // No NN exists for $r$
3  else
4      if $!Null(s_n)$ then
5          if $r.\mathbf{C} = s_n.\mathbf{C} \wedge \big(d(r, s_c) > d(r, s_n) \vee r.\mathbf{C} \neq s_c.\mathbf{C}\big)$ then
6              markPosition(InnerPlan)                   // May be first NN of $r$
7          $s_c \leftarrow s_n$
8          $s_n \leftarrow$ fetchRow(InnerPlan)
9      if $r.\mathbf{C} = s_c.\mathbf{C} \wedge \big(Null(s_n) \vee d(r, s_c) < d(r, s_n) \vee r.\mathbf{C} \neq s_n.\mathbf{C}\big)$ then
10         $s_c \leftarrow$ restorePosition(InnerPlan)       // Fetch first NN of $r$
11         $s_n \leftarrow$ fetchRow(InnerPlan)
12         $nextState = 3$                               // GoTo JoinTuples
13     else
14         $nextState = 2$                               // Last NN of $r$ not yet reched
```

5.2.3. Join Tuples

In this state we join r with all its nearest neighbors (for a given outer tuple, multiple nearest neighbors might exist). We scan the inner relation from the position marked in the state FetchInner, and we join r with s_c. If s_n has a bigger distance to r than s_c, then s_c was the last nearest neighbor of r and we go to FetchOuter. If s_n does not have a bigger distance than s_c, then s_n is also a nearest neighbor for r, and we do not change state.

State 3: JoinTuples(*NNJObj*)

```
1  $Output(r \circ s_c)$                                  // Output Result Tuple
2  if $Null(s_n) \vee d(r, s_n) > d(r, s_c) \vee r.\mathbf{C} \neq s_n.\mathbf{C}$ then
3      $nextState = 4$                                   // All NNs of $r$ are found
4  else
5      $s_c \leftarrow s_n$
6      $s_n \leftarrow$ fetchRow(InnerPlan)
7      $nextState = 3$                                   // Still NNs to fetch
```

5.2.4. Fetch Outer

In this state, we first fetch a new outer tuple. Then, in case r shares the same join matches of the previous outer tuple (lines 3-5), we go back in the inner relation to its first nearest neighbor: this is the only backtracking that our algorithm performs. We then jump to state FetchInner. In case no more outer tuples exist, the algorithm ends.

State 4: FetchOuter(*NNJObj*)

```
1 $r \leftarrow$ fetchRow(**OuterPlan**)
2 if $!Null(r)$ then
3     if $d(r, s_c) \leq d(r, s_n) \vee Null(s_n)$ then
4         restorePosition(**InnerPlan**)
5         $s_n \leftarrow$ fetchRow(**InnerPlan**)
6     $nextState = 2$                          // Search the NNs of $r$
7 else
8     $nextState = 5$                          // No more tuples to process
```

5.3. SQL Syntax Extension

Our NNJ operator can be used similar to the standard join operators, except that since our operator is category-embedded, a second argument (i.e., the category $\mathbf{C}$) must be specified. The concrete SQL syntax for $\mathbf{r} \ltimes^T [\mathbf{C}, \theta]\, \mathbf{s}$ is:

```
SELECT *
FROM r NNJ s ON T USING C
WHERE θ
```

The keyword `NNJ` specifies the join type, `ON` specifies the similarity attribute T, and `USING` the category attribute $\mathbf{C}$. Condition θ can be specified along with any other condition in the `WHERE` clause of the SQL query.

As an example consider an SQL query from the Swiss Feed Data Warehouse that computes the *derived nutrient* 'GE' (Gross Energy). The Gross Energy value of the feed samples in $\mathbf{r}$ is calculated as follows: i) one NNJ to retrieve the closest measurement of nutrient 'CP' (Crude Protein) in $\mathbf{s}$, i.e., N = 'CP'; ii) one NNJ to retrieve the closest measurement of nutrient 'OM' (Organic Matter) in $\mathbf{s}$, i.e., N = 'OM'; and iii) evaluation of the formula $0.8 * CP + 2 * OM$ on the join result.

```
SELECT C, T, 'GE', 0.8 * CP + 2 * OM
FROM (SELECT z1.*, V AS OM
      FROM (SELECT r.*, V AS CP
            FROM r NNJ s ON T USING C
            WHERE N = 'CP') AS z1 NNJ s ON T USING C
      WHERE N = 'OM') AS z2
```

Note that no materialization of intermediate join results is needed. As soon as an output tuple of $\mathbf{z}_1$ is produced, it can be pipelined to compute $\mathbf{z}_2$.

6. The *roNNJ* Query Tree

In DBMSs regular joins are category- and selection-enabled, since they support category attributes $\mathbf{C}$ and selections θ. For example, for a join $\mathbf{r} \bowtie_{\mathbf{r.C=s.C}\wedge\theta} \mathbf{s}$, each joined pair must satisfy the equality on $\mathbf{C}$ and selection θ. A category- and selection-enabled join offers two important guarantees: first, the join is computed once, independent of the number of categories in the data; second, the DBMS can take advantage of any optimizations on $\mathbf{C}$ and θ to improve performance. For example, if θ is selective, its evaluation can be pushed down before the join to reduce the number of tuples to process. Ours is the first NNJ solution that remains efficient when combined with other DBMS optimizations. Current solutions are not robust and easily degenerate with many index false hits and redundant fetches.

6.1. The roNNJ Query Tree in Row-Stores

The query tree in Figure 6(a) illustrates the base case of our approach. The top node $\ltimes^T[\mathbf{C},\theta]^{\,\mathbf{Merge}}$ of the tree is *category-* and *selection-enabled*, and represents our *roNNJ* algorithm (cf. Section 5). For each tuple of the left subtree the $\ltimes^T[\mathbf{C},\theta]^{\,\mathbf{Merge}}$ node finds in the right subtree the nearest neighbors according to T among the tuples that satisfy the equality on $\mathbf{C}$ and condition θ. The category-enabled NNJ node, $\ltimes^T[\mathbf{C},\theta]^{\,\mathbf{Merge}}$, computes its left and right subtrees only once, independent of the number of categories, and can fully leverage other optimizations.

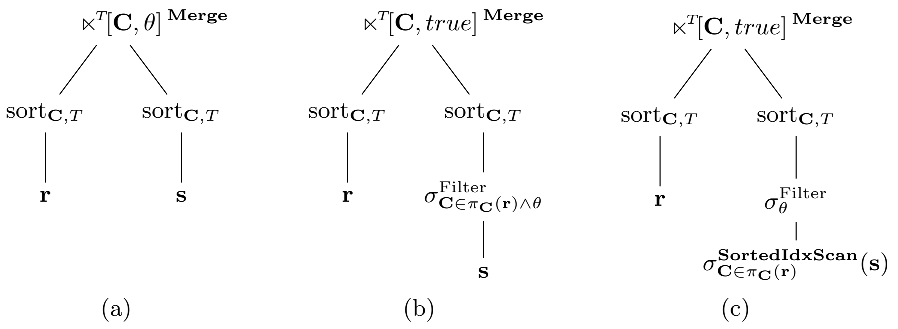

$\ltimes^T[\mathbf{C},\theta]^{\,\mathbf{Merge}}$
$\text{sort}_{\mathbf{C},T}$
$\text{sort}_{\mathbf{C},T}$
$\mathbf{r}$
$\mathbf{s}$
(a)
$\ltimes^T[\mathbf{C},true]^{\,\mathbf{Merge}}$
$\text{sort}_{\mathbf{C},T}$
$\text{sort}_{\mathbf{C},T}$
$\mathbf{r}$
$\sigma^{\text{Filter}}_{\mathbf{C}\in\pi_{\mathbf{C}}(\mathbf{r})\wedge\theta}$
$\mathbf{s}$
(b)
$\ltimes^T[\mathbf{C},true]^{\,\mathbf{Merge}}$
$\text{sort}_{\mathbf{C},T}$
$\text{sort}_{\mathbf{C},T}$
$\mathbf{r}$
$\sigma^{\text{Filter}}_{\theta}$
$\sigma^{\mathbf{SortedIdxScan}}_{\mathbf{C}\in\pi_{\mathbf{C}}(\mathbf{r})}(\mathbf{s})$
(c)

Figure 6: A Category- and Selection-Enabled Nearest Neighbor Join Tree accesses the input data only once. It can take advantage of all optimizations offered by the DBMS.

For example, the tree in Figure 6(b), prior to computing the NNJ, applies *selection pushdown* [22]. The right subtree makes an additional scan on $\mathbf{r}$ (for reading the categories stored in it) and selects from $\mathbf{s}$ only the tuples with the categories of $\mathbf{r}$. Similarly, it pushes down the evaluation of θ. The selection node passes to the $sort_{\mathbf{C},T}$ node only the portion of table $\mathbf{s}$ that contributes to the NNJ result, i.e., the tuples that have the same categories as the tuples in $\mathbf{r}$ and satisfy θ. This allows to avoid to sort all data.

In Figure 6(c) we show that, if an index on $\mathbf{s}.\mathbf{C}$ is present, the selection on $\mathbf{C}$ allows the DBMS to fetch only the blocks of $\mathbf{s}$ that are relevant to the join. The indexed selection $\sigma_{\mathbf{C}\in\pi_{\mathbf{C}}(\mathbf{r})}(\mathbf{s})$ is implemented using a *SortedIndexScan*[4] [12]. It does not fetch blocks that do not store a tuple with the categories of $\mathbf{r}$ since they will not contribute to the result. After the relevant blocks have been fetched, only the tuples with a matching category are kept. The tree in Figure 6(c) also shows how to efficiently combine the push-down of selections on θ with a SortedIndexScan on $\mathbf{C}$. In such a case, condition θ must be evaluated *before* the join, but *after* the selection on the categories $\mathbf{C}$. This allows to compute the selection on $\mathbf{C}$ using the index, i.e., without a full scan of table $\mathbf{s}$; and it allows to compute σ_θ on the fly (almost) for free when the relevant portions of $\mathbf{s}$are retrieved. Such optimizations can similarly be applied to the left subtree using the groups of $\mathbf{s}$.

Current NNJ solutions for $\ltimes^T$ are not category- and selection-enabled, which limits the scope of the query optimizer. A category-unaware NNJ will join tuples of different categories, which is wrong. Solutions based on the SegmentApply [3] operator process categories individually, and fetch for each category the blocks storing tuples of that category. If a block stores tuples of different categories, the block is fetched multiple times. This is inefficient for big datasets. Solutions based on a B-tree [4], suffer from index false hits if θ is evaluated on the fly.

6.2. The roNNJ Query Tree in Column-Stores

This subsection shows that category- and selection-enabled NNJs can also be efficiently integrated into column-stores. In column-stores, our query tree[5] combines both *early materialization* [23] and *late materialization* [24]: the former, since for computing $sort_{\mathbf{C},T}$ before the NNJ, columns[6] $\mathbf{C}$ and T need to be combined; the latter, since the rest of the columns from $\mathbf{s}$ are fetched only after the NNJ has been computed. In general, column-stores try to avoid early materialization to keep the data small. For example, in a NNJ query, for each $c \in \Pi_{\mathbf{C}}(\mathbf{r})$ they select the *oid* of the entries with category c, fetch their T value and compute the NNJ just using T. The other attributes are only fetched at the very end to construct the result tuples. Such an approach incurs redundant fetches when the relations are not clustered on $\mathbf{C}$ (remember that $\mathbf{C}$ is not fixed and changes for different NNJ queries) and makes NNJ queries slow.

Figure 7 illustrates our query tree. In the left subtree of Figure 7, first $\mathbf{r}.\mathbf{C}$ is joined to $\mathbf{r}.T$. This is not expensive because the columns of $\mathbf{C}$ and T are stored with the same *oid* order, and the join can be performed with a scan (i.e.,

[4]A sorted index scan, when all values of an IN-subquery are known up-front (i.e., $\pi_{\mathbf{C}}(\mathbf{r})$), performs for each value an index look-up on $\mathbf{s}$ and collects a list of the IDs of the (relevant) blocks storing matching tuples. It then sorts and deduplicates the list, and fetches each of the relevant blocks once in sorted order.

[5]We use MonetDB 11.21.5 [11] as a reference point.

[6]When we draw a node operating on an attribute set (e.g., $\mathbf{C} = C_1, \ldots, C_g$), the node is intended to be replicated for each attribute of the set.

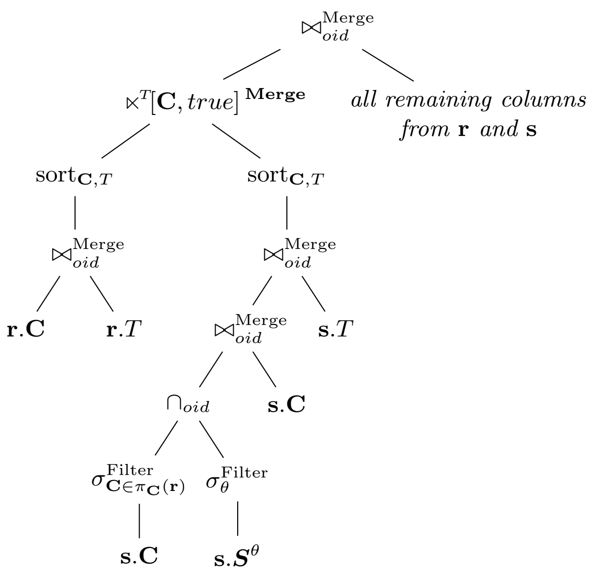

$\bowtie^{\text{Merge}}_{oid}$
$\ltimes^T[\mathbf{C}, true]^{\textbf{Merge}}$
all remaining columns from **r** *and* **s**
$\text{sort}_{\mathbf{C},T}$
$\text{sort}_{\mathbf{C},T}$
$\bowtie^{\text{Merge}}_{oid}$
$\bowtie^{\text{Merge}}_{oid}$
$\mathbf{r}.\mathbf{C}$
$\mathbf{r}.T$
$\bowtie^{\text{Merge}}_{oid}$
$\mathbf{s}.T$
$\cap_{oid}$
$\mathbf{s}.\mathbf{C}$
$\sigma^{\text{Filter}}_{\mathbf{C}\in\pi_{\mathbf{C}}(\mathbf{r})}$
$\sigma^{\text{Filter}}_{\theta}$
$\mathbf{s}.\mathbf{C}$
$\mathbf{s}.\boldsymbol{S}^{\theta}$

Figure 7: Each node in a Robust Nearest Neighbor Join Tree for column store DBMSs returns only *oid*s (e.g., the selections σ) so that the parent node can take advantage of the order with which the (*oid*, *value*) pairs are physically stored. The actual column values are returned by a join $\bowtie$ node to the parent node only when the are needed.

a merge) of two columns. Afterwards, the join result is sorted by the values of ($\mathbf{C}$,T).

In the inner subtree, $\mathbf{s}.\mathbf{C}$ is accessed and the *oid*s of the tuples with the categories of $\mathbf{r}$ are returned. Column-stores implement such a condition as an *invisible join* [25], which rewrites the semijoin between $\mathbf{r}$ and $\mathbf{s}$, as a selection predicate. It is implemented by scanning $\mathbf{s}.\mathbf{C}$ and comparing it with the entries of $\pi_{\mathbf{C}}(\mathbf{r})$, which have been stored in a hash-table. Note that if the NNJ was not category-enabled, such an optimization would be useless since, in order to ensure the correctness of the result, only one category at a time may be passed to the NNJ node. Subsequently, each column used in the θ condition (we refer to them as $\boldsymbol{S}^{\theta}$) is accessed, and the *oid*s of the tuples satisfying θ are returned. They are intersected with the *oid*s of the tuples of the selected categories. In column-stores the order of the *oid*s of a selection is preserved. Thus, the lists of *oid*s of the two selections come in the same order, and their intersection can be performed by the parent node $\cap_{oid}$ with only a scan of the two lists. Afterwards the values of $\mathbf{s}.\mathbf{C}$ and $\mathbf{s}.T$ are fetched and joined to the previous *oid*s (again, the join is performed with a scan). Finally, *roNNJ* is applied as a SortMerge procedure, and the remaining columns are concatenated to the result through a join. Thus, late materialization is used for all attributes apart from $\mathbf{C}$ and T.

Summarizing, our approach is independent of the physical layout of the relations: it avoids index false hits and redundant fetches also in column-stores. Essentially, each column is read only once.

6.3. Complexity of Query Tree

This section computes the number of operations for computing a category- and selection-enabled NNJ in terms of disk I/Os, memory I/Os and CPU operations. While current approaches tend to become quadratic when dealing with many categories or with selective predicates, we prove that our approach, independent of the category and of the predicate selectivities, is upper bounded by a complexity of $n \log n$. As usual, we exclude the cardinality of the result from our analysis.

6.3.1. Disk I/O Complexity

Lemma 2. *(Disk) Let* $\mathbf{r}$ *be a relation with schema* $\boldsymbol{R} = [\mathbf{C}, T]$*,* $\mathbf{s}$ *be a relation with schema* $\boldsymbol{S} = [\mathbf{C}, T, \mathbf{V}]$*, and* θ *be a predicate on* $\boldsymbol{S}$*. The number of disk I/Os for computing* $\mathbf{r} \ltimes^{T} [\mathbf{C}, \theta]\, \mathbf{s}$ *is independent of the category and predicate selectivities, and is upper bounded by a linearithmic complexity.*

Proof. Consider the number of Disk I/Os for computing each operation in Figure 6(b).

$$\begin{aligned}
&\mathit{Disk}(\text{read}(\mathbf{r})) = B_{\mathbf{r}}\\
&\mathit{Disk}(\text{read}(\mathbf{s})) = B_{\mathbf{s}}\\
&\mathit{Disk}(\text{sort}(\mathbf{r})) = B_{\mathbf{r}} \log_M B_{\mathbf{r}}\\
&\mathit{Disk}(\text{sort}(\sigma(\mathbf{s})) = \big(\mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}\big) \log_M \big(\mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}\big)\\
&\mathit{Disk}(\text{merge}(\mathbf{r}, \sigma(\mathbf{s})) = B_{\mathbf{r}} + \mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}.
\end{aligned}$$

$M = \mathit{max}(\mathit{mem\_size}, B)$ is the memory available for a relation with B blocks, $\mathit{sel}(\theta) = \frac{|\sigma_\theta(\mathbf{s})|}{|\mathbf{s}|}$ is the *predicate selectivity*, with $0 \le \mathit{sel}(\theta) \le 1$, and $\mathit{sel}(\mathbf{C}) = \frac{|\sigma_{\mathbf{C} \in \pi_{\mathbf{C}}(\mathbf{r})}(\mathbf{s})|}{|\mathbf{s}|}$ is the *category selectivity*, with $0 \le \mathit{sel}(\mathbf{C}) \le 1$.

As shown in Figure 6(b), the selection $\sigma(\mathbf{s})$ is evaluated on the fly while reading the $B_{\mathbf{s}}$ blocks of $\mathbf{s}$ and incurs no additional I/Os. This selection returns $\mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}$ blocks: $\mathit{sel}(\theta) * B_{\mathbf{s}}$ are needed for the tuples satisfying θ, and $\mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}$ blocks are needed for the tuples with the categories of $\mathbf{r}$. $\mathit{Disk}(\text{merge}(\mathbf{r}, \sigma(\mathbf{s})))$ is the cost of one scan done from our algorithm and it is irrelevant compared to the sorting. The dominant Disk I/O of our approach is therefore:

$$\begin{aligned}
\mathit{Disk} &\simeq \mathit{Disk}(\text{sort}(\mathbf{r}) + \mathit{Disk}(\text{read}(\mathbf{s})) + \mathit{Disk}(\text{sort}(\sigma(\mathbf{s}))\\
&= B_{\mathbf{r}} \log_M B_{\mathbf{r}} + B_{\mathbf{s}} + \big(\mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}\big) \log_M \big(\mathit{sel}(\theta) * \mathit{sel}(\mathbf{C}) * B_{\mathbf{s}}\big)
\end{aligned}$$

□

We analyze the cost for two typical scenarios. The first is the scenario when the number of query points in $\mathbf{r}$ is negligible compared to the size of $\mathbf{s}$ (e.g., in a data warehouse environment, the size of $\mathbf{r}$ is small compared to the size of the fact table $\mathbf{s}$). The second scenario is for the case when the number of query points gets large.

- *Scenario 1:* $|\mathbf{r}| \ll |\mathbf{s}|$
  From $|\mathbf{r}| \ll |\mathbf{s}|$ we get $B_\mathbf{r} \ll B_\mathbf{s}$, i.e., the cost for sorting $\mathbf{r}$ is negligible.
  1. In the best case, either θ is always false, i.e., $sel(\theta) \to 0$, or all tuples in $\mathbf{r}$ belong to the same category, i.e., $sel(\mathbf{C}) \to 0$:
     $$sel(\theta) \to 0 \vee sel(\mathbf{C}) \to 0 \Leftrightarrow Disk(\text{sort}(\sigma(\mathbf{s}))) = 0$$
     For $sel(\theta) \to 0$ *or* $sel(\mathbf{C}) \to 0$, the number of Disk I/Os on $\mathbf{s}$ provides the lower bound for our approach: since $Disk(\text{sort}(\sigma(\mathbf{s}))) = 0$, from the result of Lemma 2 we get $Disk \simeq Disk(\mathbf{s}) = B_\mathbf{s}$.
  2. In the worst case, all tuples satisfy predicate θ, i.e., $sel(\theta) = 1$, and there exists a tuple in $\mathbf{r}$ for any possible category of $\mathbf{s}$, i.e., $sel(\mathbf{C}) \to 1$.
     $$sel(\theta) \to 1 \wedge sel(\mathbf{C}) \to 1 \Leftrightarrow Disk(\text{sort}(\sigma(\mathbf{s})) = B_\mathbf{s} \log B_\mathbf{s}$$
     For $sel(\theta) \to 1$ *and* $sel(\mathbf{C}) \to 1$ the number of Disk I/Os on $\mathbf{s}$ is the upper bound of our approach: since $Disk(\text{sort}(\sigma(\mathbf{s}))) = B_\mathbf{s} \log_M B_\mathbf{s}$, we get $Disk \simeq Disk(\mathbf{s}) = B_\mathbf{s} \log_M B_\mathbf{s}$.
- *Scenario 2: otherwise.*
  Since $sel(\theta) \leq 1$ and $sel(\mathbf{C}) \leq 1$, the cost for sorting $\mathbf{r}$ dominates the cost for sorting $\sigma(\mathbf{s})$. Since $Disk(\text{sort}(\mathbf{r})) = B_\mathbf{r} \log_M B_\mathbf{r}$, we get $Disk \simeq Disk(\text{sort}(\mathbf{r})) \simeq B_\mathbf{r} \log_M B_\mathbf{r}$.

Opposite to current approaches, which suffer from false hits and redundant fetches, the number of Disk I/Os for our approach does not deteriorate in the presence of θ and $\mathbf{C}$. In fact, since $0 \leq sel(\theta) \leq 1$ and $0 \leq sel(\mathbf{C}) \leq 1$, our approach benefits from the predicate and category selectivities because the number of blocks to sort shrinks. Current solutions based on B-Tree indexes, instead, suffer since they have a $2|\mathbf{r}| \times \frac{1}{sel(\theta)}$ complexity, and, in the presence of a selective predicate, end up in fetching many blocks of $\mathbf{s}$ for each single $\mathbf{r}$ tuple. Note that the cost of our approach is independent of any clustering: we fetch each block only once independent of the number of categories stored on the block. Current solutions based on Segment Apply, instead, fetch a block m times if the block stores tuples of m categories.

6.3.2. Memory I/O Complexity

Lemma 3. *(Memory) Let $\mathbf{r}$ be a relation with schema $\boldsymbol{R} = [\mathbf{C}, T]$, $\mathbf{s}$ be a relation with schema $\boldsymbol{S} = [\mathbf{C}, T, \mathbf{V}]$, and θ be a predicate on $\boldsymbol{S}$. The number of memory I/Os for computing $\mathbf{r} \ltimes^{T} [\mathbf{C}, \theta]\, \mathbf{s}$ is independent of the category and predicate selectivities, and is upper bounded by a linearithmic complexity.*

Proof. Similar to Lemma 2. The memory I/O in the general case is:

$$\begin{aligned} Mem. &\simeq Mem.(\text{sort}(\mathbf{r})) + Mem.(\text{read}(\mathbf{s})) + Mem.(\text{sort}(\sigma(\mathbf{s}))) \\ &= B_\mathbf{r} \log_2 B_\mathbf{r} + 0 + \big(sel(\theta) * sel(\mathbf{C}) * B_\mathbf{s}\big) \log_2 \big(sel(\theta) * sel(\mathbf{C}) * B_\mathbf{s}\big) \end{aligned}$$

Since the relations are initially read from disk, there is no memory I/O for the first read of $\mathbf{s}$. For the sorting, given B blocks, $B \log_M B$ I/Os are done on disk: $B \log_2 B - B \log_M B$ I/Os are done in main memory. Since $B \log_2 B - B \log_M B = B \log_2 B * (1 - \frac{1}{log_2 M}) \simeq B \log_2 B$, the cost of sorting is given by substituting B with the number of blocks to sort, i.e., $B_{\mathbf{r}}$ for the left input, and $sel(\theta) * sel(\mathbf{C}) * B_{\mathbf{s}}$ for the right input.
□

We analyze the cost for the two typical scenarios.

- *Scenario 1:* $|\mathbf{r}| \ll |\mathbf{s}|$
  Remember that the cost for sorting $\mathbf{r}$ is negligible.
  1. In the best case:
  $$sel(\theta) \to 0 \vee sel(\mathbf{C}) \to 0 \Leftrightarrow Mem.(\text{sort}(\sigma(\mathbf{s}))) = 0$$
     For $sel(\theta) \to 0$ *or* $sel(\mathbf{C}) \to 0$, the number of Memory I/Os on $\mathbf{s}$ is the lower bound for our approach, which is zero since no sorting is computed: since $Mem.(\text{sort}(\sigma(\mathbf{s}))) = 0$, from the result of Lemma 3 we get $Mem. \simeq Mem.(\mathbf{s}) = 0$.
  2. In the worst case:
  $$sel(\theta) \to 1 \wedge sel(\mathbf{C}) \to 1 \Leftrightarrow Mem.(\text{sort}(\sigma(\mathbf{s}))) = B_{\mathbf{s}} \log_2 B_{\mathbf{s}}$$
     For $sel(\theta) \to 1$ *and* $sel(\mathbf{C}) \to 1$ the number of Memory I/Os on $\mathbf{s}$ corresponds to the upper bound of our approach: since $Mem.(\text{sort}(\sigma(\mathbf{s}))) = B_{\mathbf{s}} \log_2 B_{\mathbf{s}}$, we get $Mem. = Mem.(\mathbf{s}) = B_{\mathbf{s}} \log_2 B_{\mathbf{s}}$.
- *Scenario 2: otherwise.*
  Since $sel(\theta) \leq 1$ and $sel(\mathbf{C}) \leq 1$, then the cost for sorting $\mathbf{r}$, i.e., $Mem.(\text{sort}(\mathbf{r})) = B_{\mathbf{r}} \log_2 B_{\mathbf{r}}$, is always dominant, and we get $Mem. \simeq Mem.(\text{sort}(\mathbf{r})) = B_{\mathbf{r}} \log_2 B_{\mathbf{r}}$.

The memory I/O of our approach benefits from the predicate selectivity and the category selectivity since $0 \leq sel(\theta) \leq 1$ and $0 \leq sel(\mathbf{C}) \leq 1$. It is independent of the overlapping of the categories in the blocks. In the best case, our approach has no memory I/O at all.

6.3.3. CPU Complexity

Lemma 4. *(CPU) Let $\mathbf{r}$ be a relation with schema $\boldsymbol{R} = [\mathbf{C}, T]$, $\mathbf{s}$ be a relation with schema $\boldsymbol{S} = [\mathbf{C}, T, \mathbf{V}]$, and θ be a predicate on $\boldsymbol{S}$. The number of CPU operations for computing $\mathbf{r} \ltimes^T [\mathbf{C}, \theta]\, \mathbf{s}$ is independent on the category and predicate selectivities, and is upper bounded by a linearithmic complexity.*

Proof. (CPU) From the tree of Figure 6(b), the CPU costs of each operation are:

$$\begin{aligned}
&CPU(\sigma(\mathbf{s})) \simeq |\mathbf{s}| \\
&CPU(\text{sort}(\mathbf{r})) \simeq |\mathbf{r}| \log_2 |\mathbf{r}| \\
&CPU(\text{sort}(\sigma(\mathbf{s})) \simeq \big(sel(\theta) * sel(\mathbf{C}) * |\mathbf{s}|\big) \log_2 \big(sel(\theta) * sel(\mathbf{C}) * |\mathbf{s}|\big) \\
&CPU(\text{merge}(\mathbf{r}, \sigma(\mathbf{s}))) \simeq |\mathbf{r}| + sel(\theta) * sel(\mathbf{C}) * |\mathbf{s}|
\end{aligned}$$

For the selection $\sigma(\mathbf{s})$, each tuple of $\mathbf{s}$ must be evaluated against condition $\mathbf{C} \in \pi_{\mathbf{C}}(\mathbf{r}) \wedge \theta$. The SortMerge procedure consists of a sort and of a merge step. The merging is negligible compared to the sorting. All CPU costs are approximated (symbol $\simeq$) since each CPU operation for sorting or merging has actually cost 3: 1 for comparing the category attribute, 1 for comparing the similarity attribute, and 1 for the logical $\wedge$ between the two comparisons. The dominant CPU cost for our approach is:

$$\begin{aligned}
CPU \simeq & CPU(\text{sort}(\mathbf{r})) + CPU(\sigma(\mathbf{s})) + CPU(\text{sort}(\mathbf{s})) \\
\simeq & |\mathbf{r}| \log_2 |\mathbf{r}| + |\mathbf{s}| + \big(sel(\theta) * sel(\mathbf{C}) * |\mathbf{s}|\big) \log_2 \big(sel(\theta) * sel(\mathbf{C}) * |\mathbf{s}|\big)
\end{aligned}$$

□

We analyze the CPU cost for the two typical scenarios.

- *Scenario 1:* $|\mathbf{r}| \ll |\mathbf{s}|$
  1. In the best case:

     $$sel(\theta) \to 0 \vee sel(\mathbf{C}) \to 0 \Leftrightarrow CPU(\text{sort}(\sigma(\mathbf{s}))) \simeq 0$$

     For $sel(\theta) \to 0$ *or* $sel(\mathbf{C}) \to 0$, the number of CPU operations has its lower bound: since $CPU(\text{sort}(\sigma(\mathbf{s}))) \simeq 0$, from the result of Lemma 4 we get $CPU \simeq CPU(\mathbf{s}) \simeq |\mathbf{s}|$, which is linear.
  2. In the worst case:

     $$sel(\theta) \to 1 \wedge sel(\mathbf{C}) \to 1 \Leftrightarrow CPU(\text{sort}(\sigma(\mathbf{s}))) \simeq |\mathbf{s}| \log_2 |\mathbf{s}|$$

     For $sel(\theta) \to 1$ *and* $sel(\mathbf{C}) \to 1$ the number of CPU operations corresponds to the upper bound of our approach: since $CPU(\text{sort}(\sigma(\mathbf{s}))) = |\mathbf{s}| \log_2 |\mathbf{s}|$, we get $CPU \simeq CPU(\mathbf{s}) \simeq |\mathbf{s}| \log_2 |\mathbf{s}|$.

- *Scenario 2: otherwise.*
  Since $sel(\theta) \leq 1$ and $sel(\mathbf{C}) \leq 1$, then the cost for sorting $\mathbf{r}$ is always dominant: since $CPU(\text{sort}(\mathbf{r})) \simeq |\mathbf{r}| \log_2 |\mathbf{r}|$, we get $CPU \simeq CPU(\text{sort}(\mathbf{r})) = |\mathbf{r}| \log_2 |\mathbf{r}|$.

Example 2. We use the Swiss Feed Data Warehouse (cf. Figure 1) to compute $\mathbf{r} \ltimes^{T} [C, N = \text{‘CP’} \wedge R > 0.7]\, \mathbf{s}$ with the following parameters: 20 categories in $\mathbf{r}$ (i.e., the number of cereals); a fact table $\mathbf{s}$ with $B_{\mathbf{s}} = 10$M blocks and $|\mathbf{s}| = 1G$

tuples; the predicate selectivity $sel(N = \text{`CP'} \wedge R > 0.7)$ is 3% (i.e., among all lab analysis 3% are crude protein measurements and have a reliability greater than 0.7); the category selectivity $sel(C)$ is 5% (i.e., among all possible animal feeds 5% are cereals); *mem_size* = 4k (the default buffer size in PostgreSQL in terms of number of blocks). The size of **r** is negligible compared to the size of the fact table **s**, and from the result of Lemmas 2, 3, and 4 we get:

$$\begin{aligned} \text{Disk} &\simeq 15k \log_{4k} 15k + 10M \simeq 10M \\ \text{Mem.} &\simeq 15k \log_2 15k \simeq 200k \\ \text{CPU} &\simeq 1.5M \log_2 1.5M + 1G \simeq 1G \end{aligned}$$

Thus, the sorting has only a low impact for our approach since we take direct advantage from the predicate and the category selectivities, and reduce the amount of data to sort. This can be easily verified in our experiments in Figure 8(c) and Figure 8(d) where, respectively, for any predicate selectivity and any category selectivity below 20%, *roNNJ* stays stable.

7. Experiments

This section empirically compares our approach with the state of the art techniques for computing NNJs. We consider: *i)* the Antijoin [27] (Antijoin) that, for a pair (r,s) checks via a NOT EXISTS subquery that no closer tuple t with the same category exists in **s**; *ii)* the B-Tree [4] (B-Tree), using indexed MIN and MAX subqueries; *iii)* the SegmentApply [2],[3] (SegApply), implemented as a Lateral query running multiple (one per category) category-unaware SortMerge NNJs; *iv)* the robust NNJ (roNNJ). We compute NNJs using the Swiss Feed Data Warehouse [5]. As for our running example, we use the animal feed as category attribute and the time as similarity attribute. In the average case, we have 20 categories in **r** (i.e., the number of different cereals in the Swiss Feed Data Warehouse), $|\mathbf{r}|$ = 60k (since we have 3k timestamps per feed on average), $|\mathbf{s}|$ = 1G, a predicate with selectivity $sel(\theta) = 0.05$, and an index on the category attribute. Throughout Sections 7.1 - 7.5, we vary each of those variables and show how the approaches behave. Finally, we compute NNJs using the GREEND [28] dataset.

Due to its high runtime, the Antijoin will just be shown in scenarios where its performances are competitive. All approaches are implemented using PostgreSQL 9.3.4, Apache Cassandra 3.4, and MonetDB 11.21.5.

For the experiments on disk, we used a 2.66 GHz Intel Core i7 machine with 4GB main memory and a 480 GB Solid State Drive, running Mac OS X 10.9. The PostgreSQL cache (*shared_buffers* parameter) and the memory used for sorting (*work_mem* parameter) have been set to their default value, i.e., respectively, 32MB and 1MB.

For the experiments in main memory, we used a 2 x Intel(R) Xeon(R) CPU E5-2440 (6 cores each) @ 2.40GHz with 64GB main memory, and running CentOS 6.4 (L1 cache: 192 KB, L2 cache: 1536 KB, L3 cache: 15360 KB). The PostgreSQL cache has been set to 10 GB and the memory used for sorting is 10

GB. All indices and all data are kept in memory and no disk I/O for reading or sorting is done.

For the experiments on column-store DBMSs, we use MonetDB and Cassandra. Specifically, MonetDB has been used for the computation of SegApply (Cassandra does not support subqueries), while Cassandra for the computation of Antijoin and B-Tree (MonetDB is not robust for computing Antijoin and starved the memory of our machines; it never takes advantage of the B-Tree for computing Min/Max queries).

7.1. Scalability on Disk

Figure 8 shows that *roNNJ* is the most robust technique on disk since it fetches each block of the fact table only once, independent of the predicate and category selectivities.

The Antijoin (Figure 8(a)) quickly deteriorates since it first builds all (r, s) pairs with the same category, and then checks in $\mathbf{s}$ that no closer tuple than s exists. This has a cubic complexity, and is only efficient when the predicate (Figure 8(c)) selects very few tuples (e.g., $sel(\theta) = 10^{-6}\%$ selects only one tuple in our scenario).

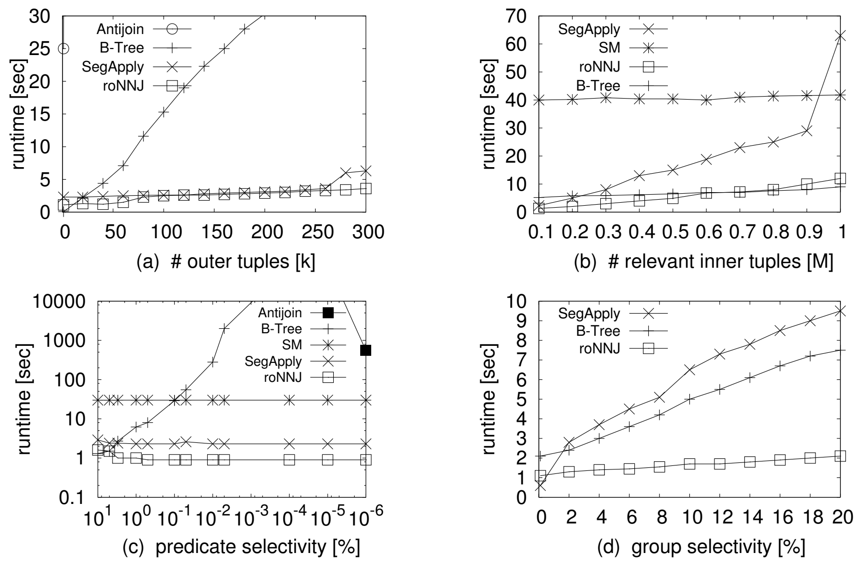


Figure 8: Scalability on Disk by varing the size of $\mathbf{r}$ (a), the size of $\mathbf{s}^{C,\theta}$ (b), the predicate selectivity (c), and the category selectivity (d).

Although the B-Tree does not require any sorting and its performances are almost independent on the size of the fact table (Figure 8(b)), it becomes extremely inefficient in the presence of a selective predicate (Figure 8(c)). Furthermore, it does not scale well if the size of $\mathbf{r}$ grows (Figure 8(a)), and becomes one order of magnitude slower than *roNNJ* for more than 200k outer tuples.

For this approach, the number of index look-ups to compute does not depend on the number of categories stored in **r**, but just on its cardinality. However, in Figure 8(d) we show that with few categories the index false hits point always to the same blocks that, once fetched, are cached in memory; with many relevant categories, instead, the false hits fetch different blocks and the runtime increases.

Figure 8(a) shows that, between 0 and 260k outer tuples, SegApply is stable since the main cost (fetching and sorting the relevant blocks of the fact table) stays the same. The jump after 260k outer tuples is because the sorting of **r** is done first in main memory using Quicksort, then on disk using external sorting. The same happens after 60k tuples for *roNNJ*, because it sorts the **r** tuples all together (and not just one segment at the time). The reader can see in Figure 8(b) that SegApply does not scale well when the size of the fact table increases, since the number of blocks to fetch redundantly increases. For more than 900k inner tuples, the blocks to read redundantly cannot all be cached in memory anymore and have to be refetched from disk. In Figure 8(d) we show that, the higher the category selectivity, the higher is the number of times that the blocks of the fact table are read redundantly.

For completeness, in Figure 8(b) and Figure 8(c), we show that non-indexed approaches (such as SortMerge, *SM*) are slower than their indexed counterpart since they fetch *all* blocks of **s** rather than just the ones storing tuples with the categories of **r**.

7.2. Scalability on Main Memory

Also in main memory, *roNNJ* is the most robust approach. For an in-memory execution, Quicksort is used by both *roNNJ* and SegApply, and no jump in the runtime occur when the size of **r** increases (Figure 9(a)). As shown in Figure 9(b), when the number of inner tuples increases SegApply reduces its gap to *roNNJ* since the latency of reading the blocks redundantly from memory is smaller than the one from disk. However it still suffers from redundant memory fetches when the category selectivity increases (Figure 9(d)).

In Figure 9(c) we show that, in the presence of an extremely selective predicate (e.g., $sel(\theta) = 0.000001$ selects only one tuple in our scenario), the Antijoin performs better than the B-Tree because it fetches from memory each relevant block $|\mathbf{r}| = 60\text{k}$ times rather than $\frac{1}{0.000001} = 1\text{M}$ times. However, the reader can see in Figure 9(d) that, opposite to the experiments on disk, the runtime of the B-Tree is pretty stable when the number of categories to process increases: it is constant up to a category selectivity of 12%, and it then slightly increases since the buffering effect of the operating systems has less impact when many categories are processed.

7.3. Scalability Without Indexes Availability

In this subsection we evaluate the approaches when no index is available on the category attribute, and we show the state of the art solutions are two order of magnitude slower than *roNNJ* since they are not category-enabled.

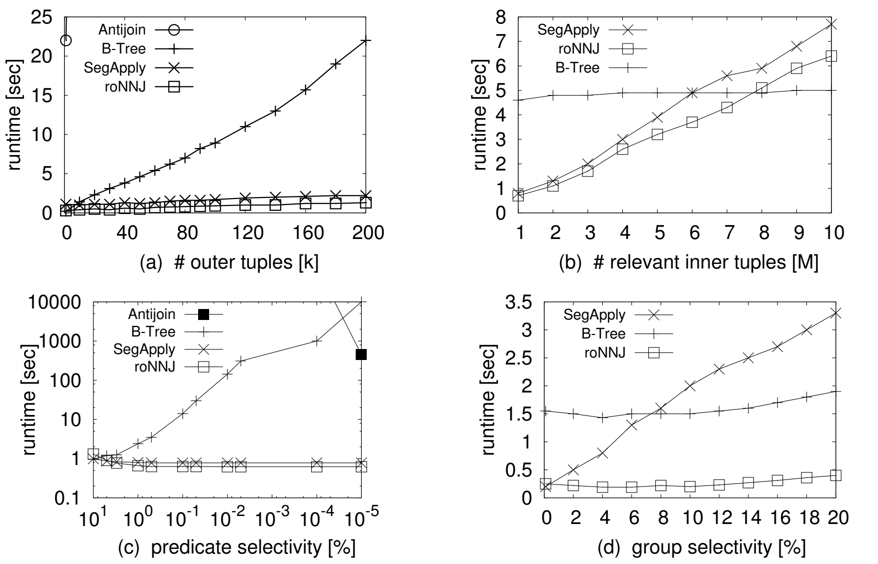


Figure 9: Scalability on Memory by varying the size of $\mathbf{r}$ (a), the size of $\mathbf{s}^{C,\theta}$ (b), the predicate selectivity (c), and the category selectivity (d).

In Figure 10(a), SegApply must performs 20 scans on the fact table (since 20 categories are stored in $\mathbf{r}$), and is thus 20 times slower than *roNNJ*. When the category selectivity increases, the number of scans of the fact table increases, too, making the approach inefficient (cf. Figure 10(b)).

When only an index on T is available, the B-Tree checks the equality on $\mathbf{C}$ similar to the selection θ. This increases the number of index false hits computed. If the closest tuple fetched does not have the same category as the outer tuple, a false hit has been computed, and the index must be scanned until a tuple with the same category satisfying θ is found. This is extremely inefficient.

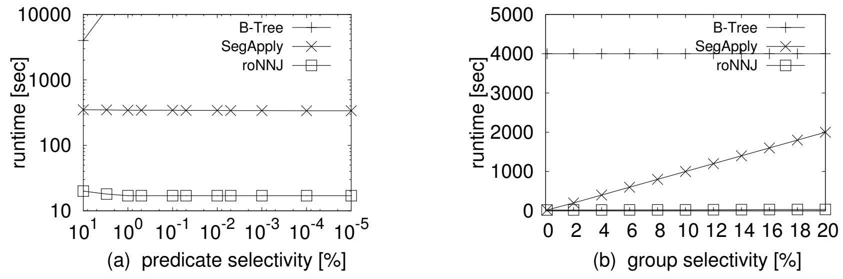


Figure 10: Scalability when an index on $\mathbf{C}$ is not available, varying the predicate selectivity (left) and category selectivity (right).

7.4. Scalability in Column-Store DBMSs

Figure 11(a) shows that the B-Tree is competitive only for a small **r** relation (e.g., less than 1k tuples). This is so because, when a secondary index is present, Cassandra finds the nearest neighbor of r by first selecting in the fact table *all* the tuples of category $r.\mathbf{C}$, and then computing the Min and Max on the selected tuples. Retrieving all the tuples of the same category makes this approach slow for a large outer relation. However, in the presence of a very selective predicate (Figure 11(c)), such an approach takes advantage of the predicate selectivity and performs opposite to its row-store counterpart. It avoids the false hits since it scans each column involved in θ and filters out all the entries not satisfying it on the fly. The approach is, at its best, 2 order of magnitude slower then *roNNJ*.

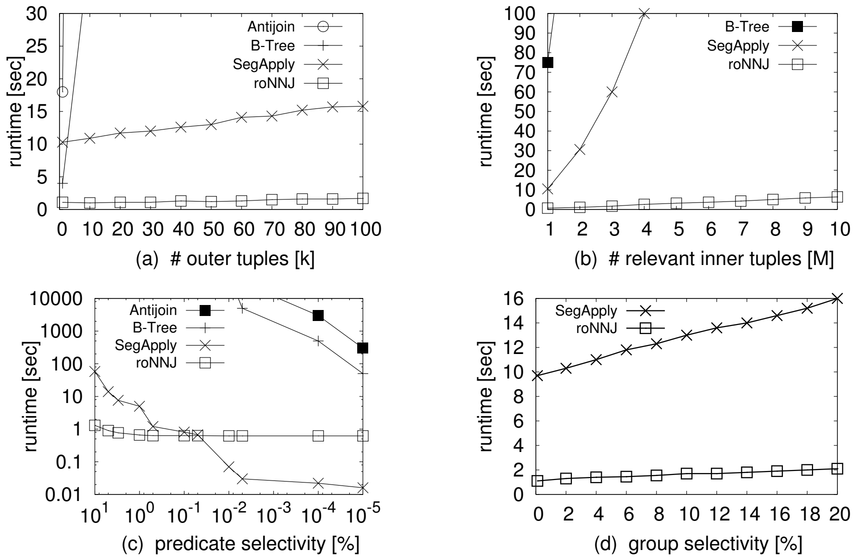


Figure 11: Scalability in Column-Stores by varying the size of **r** (a), the size of $\mathbf{s}^{C,\theta}$ (b), the predicate selectivity (c), and the category selectivity (d).

Similarly, Antijoin quickly deteriorates since it has a cubic complexity even if just **C** and T have to be accessed for computing the distances.

By comparing Figure 11(a) with Figure 9(a), the reader can see that SegApply suffers much more when it is implemented in column-store DBMSs because every column accessed before the NNJ (i.e., **C**, T, and the ones involved in θ) is affected by redundant fetches. In our experiments redundant fetches are repeated 3 times: for the category attribute (*feed_name*) and for the attributes involved in θ (*nutrient_name* and *reliability*). Furthermore, by comparing Figure 11(b) with Figure 9(b), the reader can see that, since a block stores much more (OID, Value) pairs than tuples, the probability that a block stores data of different categories is much higher in column-store than in row-store databases, and much more redundant fetches are computed. However, in the presence of a very selective predicate (Figure 11(c)), SegApply by applying early materialization fetches T only for the (few) entries satisfying θ, and speeds up the NNJ.

7.5. Scalability on a Clustered Fact Table

In this subsection we evaluate the approaches in the atypical scenario when a primary index is available, i.e., when the fact table is clustered by $(\mathbf{C}, T)$. This cannot always be ensured in real world applications, since the category and similarity attributes change for different queries. In Figure 12(a) we show that when a primary index is available, the B-Tree suffers less compared to Figure 8(a) because the data is clustered: each index false hit happens on the same or on the next block that, once fetched, is cached in main memory. The approach[7] remains however not competitive compared to *roNNJ*. SegApply performs the same as *roNNJ* since no redundant fetches are computed (most of the blocks store tuples of exactly one category). SegApply implemented on column-store DBMSs is even faster than *roNNJ*, for two reasons: *i)* when the fact table is clustered by $(\mathbf{C}, T)$ no redundant fetches are done; *ii)* the value of *every* attribute (except T) is fetched after the NNJ in sort-order, i.e., with just a scan of the columns.

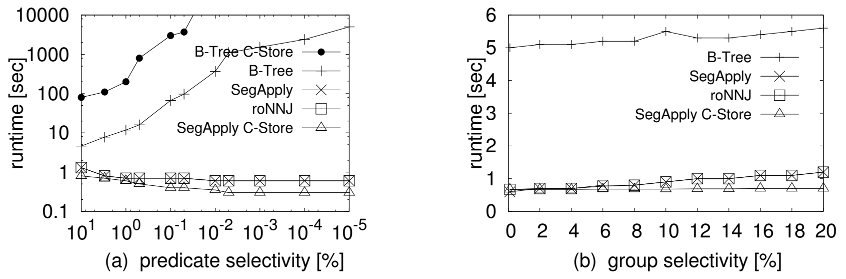


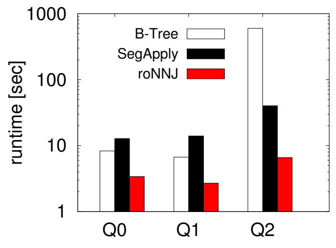
Figure 12: Scalability on a Clustered Fact Table, by varying the predicate selectivity (left) and category selectivity (right).

7.6. Real World Queries in the Swiss Feed Data Warehouse

In this subsection we evaluate how the approaches compute three queries Q0, Q1, and Q2 computing *derived nutrients* in the Swiss Feed Data Warehouse. Derived nutrients are computed thorugh a sequence of NNJs (cf. Section 5.3):

[7]For the experiment on column-store DBMSs, we use a clustered B-Tree in Apache Cassandra, and we rewrite the Min(T)/Max(T) queries as ORDER BY T LIMIT 1 statements. MonetDB does not optimize MIN(T)/MAX(T) queries when a predicate θ is present, even if a primary index on T is available. It is therefore less efficient than Cassandra in computing NNJs with a B-Tree since, for a given $r \in \mathbf{r}$, it always fetches all tuples from $\mathbf{s}$ with category $r.\mathbf{C}$.

Q0, with selectivity $sel(\theta) = 0.1$ and with two NNJs, calculates the *Gross Energy* value; Q1, with $sel(\theta) = 0.05$ and with two NNJs calculates the *Degradability of Proteins*; Q2, with $sel(\theta) = 0.1$ and with five NNJs, calculates the *Absorbable Proteins*. For each of those queries, 20k **r** tuples of 3 different feeds (categories) have been used; the predicate θ is not important since, independent of the condition itself, only its selectivity $sel(\theta)$ influences the runtime (in row-store DBMSs) of the approaches.



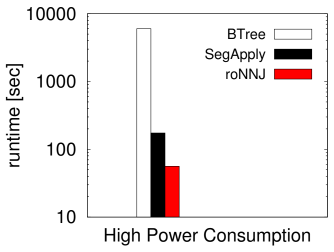


Figure 13: *Left* Chart: Top Queries in Swiss Feed Data Warehouse: Q0 ($sel(\theta) = 0.1$, 2 NNJs), Q1 ($sel(\theta)$=0.05, 2 NNJs), Q2 ($sel(\theta)$=0.1, 5 NNJs). *Right* chart: Query on the GREEND dataset: High Power Consumption ($sel(\theta) = 0.001$,1 NNJ)

The left plot of Figure 13 shows that Q2 is the query taking longest among the three. This is due to the higher number of joins (5 NNJs) computed. The B-Tree is two order of magnitude slower than *roNNJ* since, after each NNJ, the number of outer tuples for the next NNJ of the sequence gets bigger due to multiple join matches: the number of index look-ups to compute compared to the previous join also grows. Comparing Q0 with Q1 (same number of NNJs but $sel(\theta)$ reduced from 0.1 to 0.05), we see that the B-Tree and *roNNJ* become slightly better: the former since the number of result tuples of the first join decreases (this determines the number of outer tuples for the second join of the sequence), the latter since the number of relevant tuples to sort shrinks. Overall *roNNJ* is the fastest for all three queries because it does not fetch blocks redundantly and it does not suffer from θ.

7.7. Real World Query for the GREEND dataset

The GREEND dataset [28] is publicly available and contains detailed power usage information obtained through a measurement campaign in households in Austria and Italy from January 2010 to October 2014. It stores 110M measurements. Tuples are stored as $< BuildingId, T, Device_1, \ldots, Device_m >$ where T is the time when the current measurement has been taken at the building identified by $BuildingId$, and $Device_i$ stores the amount of energy consumption of a given device. The right plot of Figure13 shows how the approaches perform on the GREEND dataset for the following query:

High power usage: For each measurement where $Device_1$ is switched on, compare its power consumption with the power consumption when all devices are on:

- $\mathbf{C} = \{buildingID\}$, $T = T$
- $|\mathbf{r}| = 11\text{M}$, i.e., the rows where $Device_1$ is larger than 0
- $sel(\theta) = 1/1000$, i.e., $\theta \equiv Device_1 > 0 \wedge \ldots \wedge Device_n > 0$ selects 0.1% of the tuples
- $|\pi_{buildingID}(\mathbf{r})| = 9 \Rightarrow sel(\mathbf{C}) = 1$, i.e., 9 buildings have been monitored for this dataset, and for each of them measurements exist where the $Device_1$ is on.

The B-Tree performs worst because of a highly selective θ and a high number of $\mathbf{r}$ tuples: 11M $\times 2 \times 1000$ index false hits are computed. SegmentApply performs also slower than *roNNJ* because of the redundant fetches: since only 9 categories are present, indices cannot help because the number of tuples of the same category is too high, and 9 scans (i.e., one per category) of the dataset are computed. *roNNJ* performs best because it access the dataset only once and does not compute index false hits.

8. Conclusion and Future Work

In this work we have introduced a new algebraic operator: the category- and selection-enabled Nearest Neighbor Join. Its evaluation query tree is not dependent on the physical organization of the relations, and, opposite to the state of the art solutions, does not suffer from index false hits and redundant fetches. We have described the implementation of our query tree both in row-store and in column-store DBMSs. We have shown that, opposite to the state of the art solutions, a category- and selection-enabled query tree enlarges the scope of the query optimizer, which can take full advantage of the optimizations on the categories and on the predicate. We have implemented an efficient algorithm, *roNNJ*, that computes the NNJ in a single scan of the input relations. We have analytically shown that our approach is upper bounded by a complexity of $n \log n$. As future work, we intend to introduce a NNJ that computes the similarity for timestamps with different granularities: in the Swiss Feed Data Warehouse, for some measurements, only the month, the season or the year are available instead of the full date. We also intend to apply our findings in nearest neighbor joins with queries with user-defined distance functions.

Acknowledgments

This work has been done in collaboration with Agroscope Switzerland in the context of the Tameus project, with funding from the Swiss National Science Foundation. We thank the anonymous reviewers for their insightful suggestions.

9. References

[1] Silva, Y.N., Aref, W.G., Larson, P.Å., Pearson, S., Ali, M.H.: Similarity queries: their conceptual evaluation, trans- formations, and processing. VLDB J. **22**(3) (2013) 395–420

[2] Silva, Y.N., Aref, W.G., Ali, M.H.: The similarity join database operator. In: ICDE. (2010) 892–903

[3] Galindo-Legaria, C.A., Joshi, M.: Orthogonal optimization of subqueries and aggregation. In: SIGMOD. (2001) 571–581

[4] Yao, B., Li, F., Kumar, P.: K nearest neighbor queries and knn-joins in large relational databases (almost) for free. ICDE **0** (2010) 4–15

[5] Taliun, A., Böhlen, M., Bracher, A., Cafagna, F.: A gis-based data analysis platform for analyzing the time-varying quality of animal feed and its impact on the environment. In: iEMSs. (2012)

[6] TPC: TCP-H benchmark. `http://www.tpc.org/tpch/` (2015)

[7] Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. p.269-271, 2nd edn. John Wiley & Sons, Inc. (2002)

[8] Vassiliadis, P.: Encyclopedia of Database Systems, p. 671. Springer US (2009)

[9] Cafagna, F., Böhlen, M.H., Bracher, A.: Nearest neighbour join with groups and predicates. DOLAP (2015) 39–48

[10] Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2) (2010) 35–40

[11] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: Monetdb: Two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1) (2012) 40–45

[12] Kimura, H., Huo, G., Rasin, A., Madden, S., Zdonik, S.B.: Correlation maps: A compressed access method for exploiting soft functional dependencies. PVLDB **2**(1) (2009) 1222–1233

[13] Soliman, M.A., Antova, L., Raghavan, V., El-Helw, A., Gu, Z., Shen, E., Caragea, G.C., Garcia-Alvarado, C., Rahman, F., Petropoulos, M., Waas, F., Narayanan, S., Krikellas, K., Baldwin, R.: Orca: a modular query optimizer architecture for big data. In: SIGMOD. (2014) 337–348

[14] Antova, L., El-Helw, A., Soliman, M.A., Gu, Z., Petropoulos, M., Waas, F.: Optimizing queries over partitioned tables in mpp systems. In: SIGMOD. (2014) 373–384

[15] Böhm, C., Krebs, F.: Supporting kdd applications by the k-nearest neighbor join. In: DEXA. (2003) 504–516

[16] Pagh, R., Pham, N., Silvestri, F., Stöckel, M.: I/o-efficient similarity join. In: ESA. (2015) 941–952

[17] Emrich, T., Kriegel, H.P., Kröger, P., Niedermayer, J., Renz, M., Züfle, A.: On reverse-k-nearest-neighbor joins. GeoInformatica **19**(2) (2015) 299–330

[18] Aly, A.M., Aref, W.G., Ouzzani, M.: Spatial queries with two knn predicates. PVLDB **5**(11) (2012) 1100–1111

[19] Jacox, E.H., Samet, H.: Metric space similarity joins. ACM Trans. Database Syst. **33**(2) (2008) 7:1–7:38

[20] Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: Similarity join in metric spaces. In: ECIR. (2003) 452–467

[21] Dohnal, V., Gennaro, C., Zezula, P.: Similarity join in metric spaces using ed-index. In: DEXA. (2003) 484–493

[22] Ullman, J.D.: Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. (1990)

[23] Harizopoulos, S., Liang, V., Abadi, D.J., Madden, S.: Performance tradeoffs in read-optimized databases. VLDB (2006) 487–498

[24] Abadi, D.J., Myers, D.S., DeWitt, D.J., Madden, S.R.: Materialization strategies in a column-oriented dbms. In: ICDE. (2007) 466–475

[25] Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: How different are they really? SIGMOD (2008) 967–980

[26] Blasgen, M.W., Eswaran, K.P.: Storage and access in relational data bases. IBM Syst. J. **16**(4) (1977) 363–377

[27] Mishra, P., Eich, M.H.: Join processing in relational databases. ACM Comput. Surv. **24**(1) (1992) 63–113

[28] Monacchi, A., Egarter, D., Elmenreich, W., D'Alessandro, S., Tonello, A.M.: GREEND: an energy consumption dataset of households in Italy and Austria. In: SmartGridComm. (2014) 511–516