Z-KNN Join for the Swiss Feed Database: a feasibility study

Francesco Luminati

University Of Zurich, Switzerland

francesco.luminati@uzh.ch

1 Introduction

K-nearest neighbor query (knn) and k-nearest neighbor join (knnj) are becoming fundamental in many applications, such as Knowledge Discovery in Databases, spatial databases and so on. Many previous approaches require spatial indices or changes in the database engine, posing a big limitation to the implementation. Moreover, the SQL approach normally relies on a User Defined Function (UDF), such as the Euclidean distance, which is not optimized by the query optimizer.

The authors of [1] designed an algorithm that can be implemented using only primitive SQL, exploiting so the standard functionalities of a DBMS, such as the query optimizer. In this report we test the approach against the Swiss Feed Database, a vertically organized database containing millions of measurements of nutrients in animal food.

2 K-nearest neighbor join

The k-nearest neighbor join (k-nnj) can be defined as the operation in which each point of the vector set $Q = \{q_1, ..., q_n\}$ must be combined with its k closest partners in vector set $R = \{r_1, ...r_n\}$. Formally it is defined as follow:

Definition: k-nn Join $Q \underset{k-nn}{\ltimes} R$

$Q \underset{k-nn}{\ltimes} R$ is the smallest subset of $Q \times R$ that contains for each point of Q at least k points of R and for which the following condition holds:

$$\forall (q, r) \in Q \underset{k-nn}{\ltimes} R, \forall (q, r') \in Q \times R \setminus Q \underset{k-nn}{\ltimes} R : \|q - r\| < \|q - r'\|$$

Here $\|\cdot\|$ denotes the distance between two points, usually corresponding to the Euclidean distance for multidimensional vector spaces. The SQL query for solving the k-nnj problem can be expressed as follow [2]:

```
SELECT * FROM Q,
    (SELECT * FROM R
        ORDER BY ||Q.obj - R.obj||
        LIMIT k)
```

We immediately notice that this query is going to order the table R by distance for *every* point in Q. This means that the query gives very bad performance, especially with big tables. Furthermore, to calculate the $\|\cdot\|$ a User Defined Function (UDF) is needed to compute the Euclidean distance between two points. Because a UDF is not a native part of the DBMS, the query optimizer is not able to optimize the execution plan of the query and ends up to plan an expensive nested-loop join [1].

3 Efficient algorithm

In [1] an efficient algorithm is presented. It can be implemented using only primitive SQL operators and do not rely on the UDF as a main query condition, so that the query optimizer can understand and optimize the query. The paper shows an approximate and an exact k-nn algorithm for one query point q and an extension of it to the k-nnj. The approximate algorithm gives a constant factor approximation with only O(log N) page accesses in any fixed dimension, the exact algorithm can achieve similar efficiency for certain types of data distribution.

For the scope of this report we are going to investigate in detail only the approximate algorithm in two dimensions, because the implementation of it in one dimension already returns the exact result, as we will see later.

3.1 Z-knn approximate algorithm

The idea behind the approximate Z-knn algorithm is to reduce the multidimensionality to one dimension, and then do a range scan around q over one dimension, using a B+tree index. If the locality of the points in one dimension is maintained, the nearest neighbors in multi-dimensions are going to be near q also in one dimension, making them easy and fast to find using an index.

Z-value

To map the multidimensionality to one dimension, the Authors choose the Z-value function, which is calculated by interleaving the binary representation of the coordinates X, Y from the most significant bit (msb) to the least significant bit (lsb) (see Example 3.1).

Example 3.1: calculate the Z-value of coordinate (4, 6):
Y: 6 -> *binary:* 110
X: 4 -> *binary:* 100
Z-value: 111000 -> *decimal:* 56

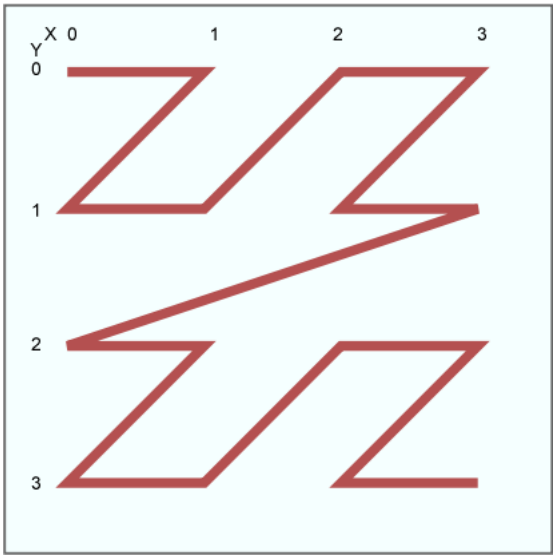

Figure 1: Z-value curve: the Z-shaped curve is produced connecting the Z-values of the coordinates.

The computation of Z-values only requires bit-shift operations, which are already available in most database engines [1]. The Z-value curve, as seen in Figure 1, is obtained by connecting the points by the numerical order of their Z-value, producing the characteristic Z-shaped curve. In most cases, Z-values preserve the locality of points and we can find the neighbors of q in a close neighborhood on the Z-value curve, lets say γ position up and down. However it is not always the case, as we will see later in the example. To get a theoretical guarantee, we create α shifted copies of the table using α random vectors $\vec{v}$. The operation is defined as shifting all points in R^i, $\forall_i \in [1, \alpha]$, by the vector $\vec{v_i}$ doing $r + \vec{v_i}$ for all $r \in R^i$.

Algorithm

For the scope of the algorithm, we add to every table R^i a column named *Z-value* containing the Z-value of the respective point and we sort the table, createing a B+tree index on the Z-value column. This operation is executed in a pre-processing phase, because the same tables and index can be reused for every query by the algorithm.

Now, for a query point q and its Z-value z_q, we look for the successor of z_q among all Z-values in R^i, and define it as z_r ($z_r >= z_q$). The γ-neighborhood of q is defined as γ points up and γ points down to z_r. At the end, the algorithm, as shown in Algorithm 1, selects the top k points in the unioned $(\alpha + 1)$ γ-neighborhood, using the Euclidean UDF to calculate the distances.

Example

The algorithm is now explained using an example to highlight some important facts. The original table R^0 is rappresented in Table 1 and we produce one shifted copy of it ($\alpha = 1$)

Algorithm 1 approximative Z-knn (query point q, point sets $\{R^0, \ldots, R^\alpha\}$)

1. Candidates $C = \emptyset$;
2. **for** $i = 0, \ldots, \alpha$ **do**
3. Find z_r^i as the successor of $z_{q+\vec{v_i}}$ in R^i;
4. Let C^i be γ points up and down next to z_r^i in R^i;
5. For each point r in C^i, let $r = r - \vec{v_i}$;
6. $C = C \cup C^i$;
7. Let A = knn(q, C) and output A.

knn(q, C) selects the top k points (nearest neighbors) using the Euclidean UDF.

Z-val	Pid	X	Y
1	1	1	0
2	2	0	1
3	3	1	1
6	4	2	1
14	5	2	3
15	6	3	3
16	7	4	0
17	8	5	0
26	9	4	3
33	10	1	4
35	11	1	5
36	12	2	4
37	13	3	4
38	14	2	5
44	15	2	6
57	16	5	6

$\xrightarrow{\vec{v_1} = (2,1)}$

Z-val	Pid	X	Y
7	1	3	1
12	2	2	2
13	3	3	2
22	7	6	1
23	8	7	1
24	4	4	2
39	10	3	5
45	11	3	6
48	5	4	4
49	6	5	4
50	12	4	5
51	13	5	5
52	9	6	4
56	14	4	6
58	15	4	7
63	16	7	7

Table 1: R^0 (left) and R^1 (right), shifted usign vector $\vec{v_1} = (2, 1)$.

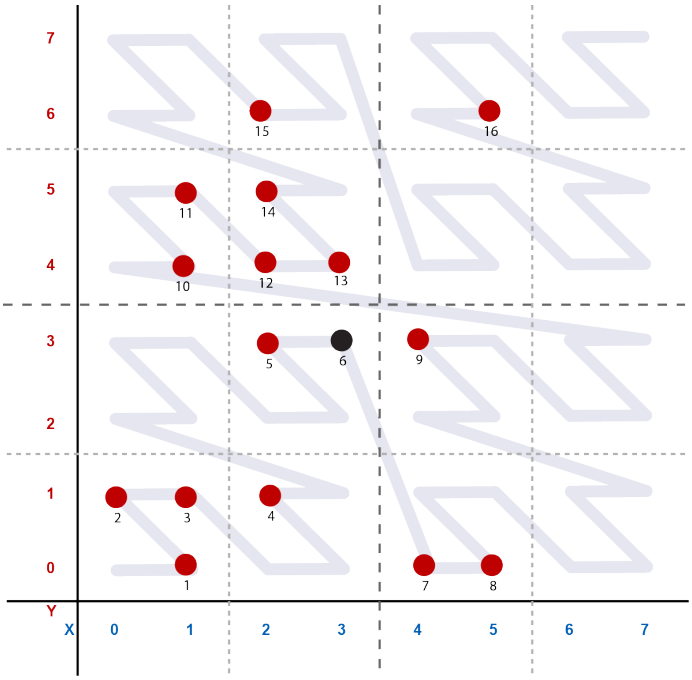

Figure 2: Cartesian representation of points in table R^0, overlapping the Z-value curve.

using the vector $\vec{v_1} = (2, 1)$. The parameters in the example are: $\gamma = 3$ and $k = 3$, the query point used is $q = (3, 3)$, $z_q = 15$. Again, γ is the number of point selected in the scan up and down of q on the Z-value curve and k is the number of neighbours we are interested to find.

To enforce the understanding of the example we propose Figure 2 and Figure 3, where the points of table R^0 and R^1 are drawn in a Cartesian coordinate system, with underlying the Z-value curve.

At this point we apply the approximative Z-knn algorithm to both table R^0 and R^1: the successor of $z_q = 15$ in R^0 is $z_r^0 = 15$, which correspond to the *pid* 6; the successor of $z_{q+\vec{v_1}} = 49$ in R^1 is $z_r^1 = 49$ (*pid* 6). The scan of γ points up and down from z_r^0 in table R^0 produces the candidates $C^0 = pid\{3, 4, 5, 7, 8, 9\}$, the scan on table R^1 the candidates $C^1 = pid\{5, 9, 10, 11, 12, 13\}$. From line 6 of the algorithm, $C = C^0 \cup C^1 = pid\{3, 4, 5, 7, 8, 9, 10, 11, 12, 13\}$ and the final k-nearest neighbors knn(q, C) are $pid\{5, 9, 13\}$.

The locality issue of Z-value curve is evident in R^0 and Figure 2. The simple γ-scan over Z-values on just R^0 would have missed two "good" candidates, *pid* 12 and 13. Only thanks to the shifted table copy R^1 the algorithm has found also these points, returning the correct answer.

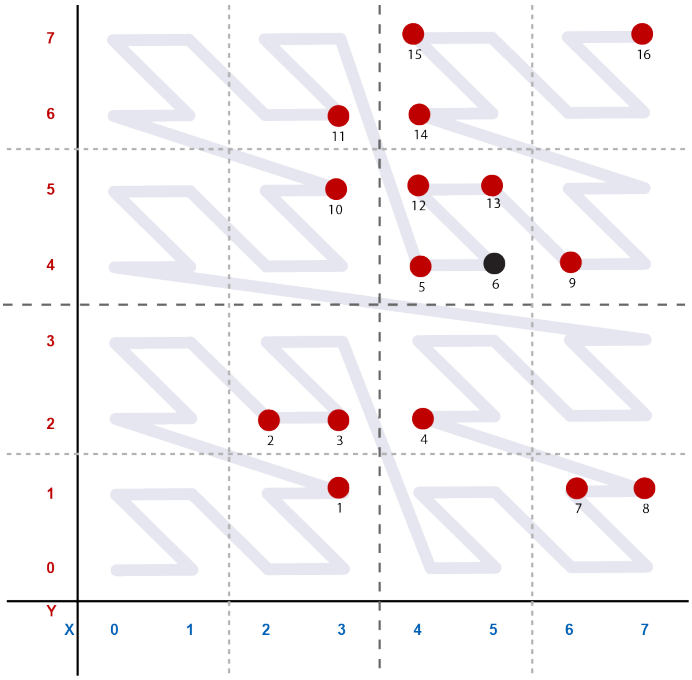

Figure 3: Cartesian representation of points in table R^1, overlapping the Z-value curve.

4 Z-knn algorithm in one dimension

In a one-dimensional space things are simpler. There is no need to map multidimensionality to one dimension, so the Z-value transformation is not necessary and the issues of this transformation fall. This means that we can create the index directly on the interested column, for example a timestamp, without the need of the Z-value column and also the shifted copies of the table are not necessary anymore. The one-dimensional situation implies that a γ scan up and down of q will immediately detect all the exact neighbors. Moreover, we can do some assumption: q can be used directly to find its successor r_q in R, $\gamma = k$ already ensure to find the k-nearest neighbors and the Euclidean UDF in one-dimension results to be a simple subtraction $UDF() = \|q - r\|$.

4.1 The Swiss Feed Database

The Swiss Feed Database contains chemical parameters of 155 nutrients about more than 600 animal feed types. Companies, farmers and research institutions use this data to preserve healthy, effective and cheap animal feed. The chemical parameters of the nutrients are elaborated in two ways: the first implies chemical analyses on feed samples; the second calculate parameters using the laboratory measurements of other nutrients.

These calculations are well-known dependencies between nutrients and are computed with the help of regressions. The complexity of the regression depends on the number of involved nutrients and spreads from one to many. The results of the regressions are called derived attributes and can be defined as functions depending on other nutrients and on

time.

The calculation of derived attributes encounter the problem that for every timestamps only a few chemical laboratory measurement are performed for an economical question. This means that the function to compute the derived attributes for every timestamp has to find the nearest neighbors of all the parameters involved that are not available.

Moreover the neighbor search is applied on a non-fixed subset of data that changes based on user selections, making the derived attributes not pre-computable.

4.2 Z-knn Join

For the computation of derived attributes we frequently need to join the fact table with its nearest neighbors, so we have extended the k-nnj SQL query seen in Section 2 to compute the one-dimensional Z-knn algorithm, producing the Z-knnj.

Now the Z-kknj is applied to the Swiss Feed Database to calculate the derived attributes *Vit-A*, defined by $f(\frac{Ca*Na}{100})$. In the Swiss Feed Database, the data are very sparse, so that a vertical design of the table has been chosen to spare space, having every foods and nutrients in one fact table.

```
SELECT * FROM fact_table AS RQ, (
   ((((SELECT * FROM fact_table AS RP WHERE RP.timestamp >= RQ.timestamp
                  AND nutrient = 'Ca' AND food = 'Hay'
                        ORDER BY (RP.timestamp) ASC LIMIT 1)
         UNION
   (SELECT * FROM fact_table AS RP WHERE RP.timestamp < RQ.timestamp
                  AND nutrient = 'Ca' AND food = 'Hay'
                        ORDER BY (RP.obj) DESC LIMIT 1)) AS RA
ORDER BY ABS(RA.timestamp-RQ.timestamp) LIMIT 1) * (
   ((SELECT * FROM fact_table AS RP WHERE RP.timestamp >= RQ.timestamp
                  AND nutrient = 'Na' AND food = 'Hay'
                        ORDER BY (RP.timestamp) ASC LIMIT 1)
         UNION
   (SELECT * FROM fact_table AS RP WHERE RP.timestamp < RQ.timestamp
                  AND nutrient = 'Na' AND food = 'Hay'
                        ORDER BY (RP.obj) DESC LIMIT 1)) AS RA
ORDER BY ABS(RA.timestamp-RQ.timestamp) LIMIT 1)) / 100) AS Vit-A
WHERE nutrient IN ('Ca', 'Na')
```

What the query does is first to limit the fact_table to the elaboration of only one food, the *Hay*. Then, the nearest neighbor of the nutrient *Ca* and *Na* is found, and the *Vit-A* is calculated using the formula $f(\frac{Ca*Na}{100})$. All this results are then joined to all the timestamps of the specific food in the fact_table.

4.3 Working dataset

Here we expose a situation where the Z-knnj algorithm can express the maximum of efficiency. Suppose there is a table for every nutrient containing the measurements and timestamps. The interested derived attribute *Vit-A* is defined by $f(\frac{Ca*Na}{100})$. Starting with timestamp 8, let say that the *Ca* value for timestamp 8 is found in the *Ca* table

Timestamp	Quantity
2	80
5	86
7	75
8	70
10	68
12	72
18	81
20	85

Timestamp	Quantity
1	3
3	2.5
4	3.5
6	3.6
9	5
11	2
15	4.5
16	2.5

Table 2: Table of the *Ca* nutrient measurements (left) and the *Na* nutrient mesurements (right).

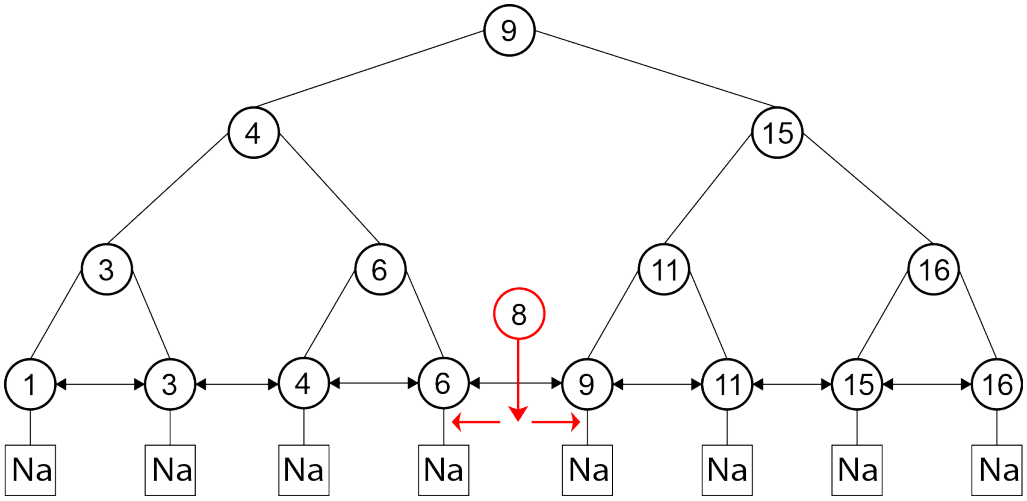

Figure 4: B-tree of the Na table, the neighbors of timestamp 8 are found immediately.

(Table 2), but there is no *Na* value for that timestamp. A Z-knn search on the *Na* table (Table 2) immediately produce two candidates, timestamp 6 and 9, as seen in the B-tree on Figure 4. At the end the algorithm will return timestamp 9 as the nearest neighbor of 8.

4.4 Not working dataset

The situation in the Swiss Feed Database is quite different as we know. The data are vertically organized, having every foods and nutrients in one fact table. Let say the table is organized in a timestamp, food, nutrient and quantity column (Table 3). To calculate the derived attribute *Vit-A* ($f(\frac{Ca*Na}{100})$) for the food *Hay,* starting by timestamp 9, we get the *Ca* value. Now the Z-knn algorithm is looking for the *Na* neighbors for food *Hay* of timestamp 9, finding them only 4 leaves left and 3 leaves right in the B-tree (Figure 5). Of course the algorithm is working, returning the value at timestamp 16, but its execution is much less performing, requiring more page accesses. The join query in Subsection 4.2

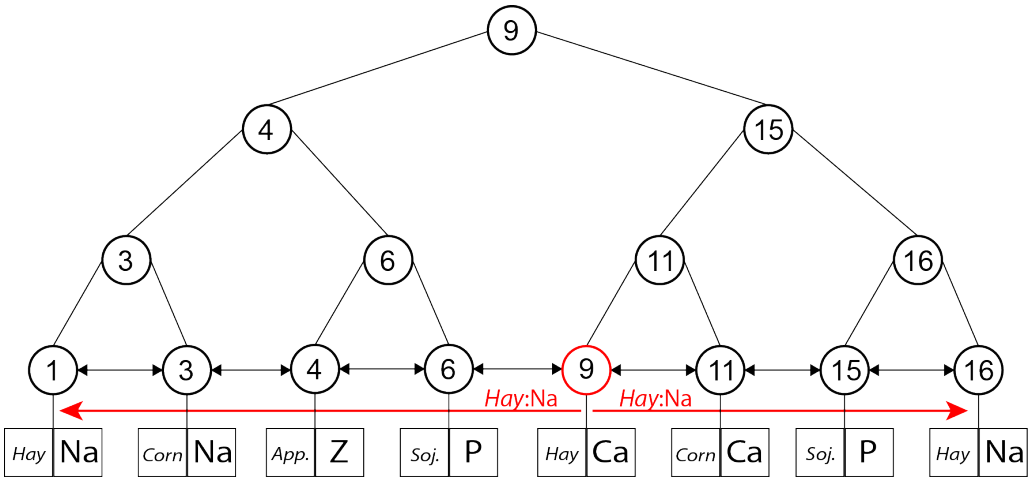

Figure 5: B-tree of the fact table, finding the neighbors here can degrade to a range search.

must be extended with the instruction WHERE food = 'Hay' AND *nutrient = 'Na'*, as we have seen. We immediately notice that the additional condition severely restrict the table; the B-tree is used to find the starting point q (in this case timestamp 9), but then if the neighbors are not on the same leaf or nearby it, a range scan left and right q must be done, degrading the performance and nullifying all the benefits of the algorithm.

5 Conclusion

The Z-knn algorithm in one-dimension is simple and fast if the dataset behind R is dense, i.e. it contains many frequently occurring items, and is not severely restricted by additional conditions. In this case, the B+tree can give the best to locate q and scan the neighbors.

This is not the case of the Swiss Feed Database, where we need the nearest neighbors of only a minimal part of the table and the data are very sparse. The additional conditions in the query causes the degradation to a range scan, loosing every benefit of the algorithm and the B+tree. Therefore the Z-knn algorithm results to be of no interest for the Swiss Feed Database. In fact, the SQL query in Subsection 4.2 has been tested on the Swiss Feed Database, giving the expected result, namely that the query takes hours to complete.

Timestamp	Food	Nutrient	Quantity
1	Hay	Na	4
3	Corn	Na	2.5
4	Apple	Z	1.1
6	Soja	P	3
9	Hay	Ca	80
11	Corn	Ca	70
15	Soja	P	6
16	Hay	Na	2.3

Table 3: Vertically organized fact table.

References

[1] Byn Yao, Feifei Li, Piyush Kumar: K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free. Data Engineering (ICDE) Conference. (2010)

[2] Christian Böhm, Florian Krebs: The k-nearest neighbour Join: Turbo Charging the KDD Process. Knowledge and Information Systems. (2004)