The Zizo® Pattern Database Technology

June 2015

by

Tom Longshaw

© Zizo Software Ltd. 2015

This paper provides technical detail as to how the Zizo Database works. It describes the nature of pattern databases and presents some important results about the performance parameters of this class of database that make it very compelling for use with large data sets.

Contents

The Zizo® Pattern Database Technology 1
- Introduction 3
- The Pattern Database 3
- Partition 4
- The Associative Store 8
- Conclusions 9
- References 9

Zizo® and Pathway™ are trademarks of Zizo Software Limited.

© Zizo Software Ltd. 2015

Introduction

In this paper I will provide a technical presentation of the capabilities and performance that can be expected from the Zizo database. We will establish that performance for SQL queries with Zizo approaches $O(\log(p))$ where p is the size of the pattern space. This makes Zizo performance comparable to the performance expected of a typical NoSQL database.

The Pattern Database

A table in the Zizo pattern database consists of a triple <*I,M,P*> where *I* is an index, *M* is the metadata, *P* is the pattern space. The index identifies which patterns are in the table, the metadata identifies the types etc. that provide meaning to the patterns, and *P* is simply an indexed collection of patterns.

A pattern can be any data object in the system. Typical patterns are the literal values that appear in columns in the data. Patterns can also be compound patterns consisting of two or more sub-patterns. Even an index can be a pattern, so at some points in the computation it can act as an index, but at others it is simply a pattern. Every pattern is annotated with three integer values which are:

- the age of the pattern (in milliseconds)
- the cost to compute/load the pattern (also measured in milliseconds)
- a reference count for the pattern (the number of times the pattern is referred to in the pattern space)

We shall return to the function of these values later.

A Zizo table is created initially by the "builder" process which analyses the columns in the table and tokenises them replacing each pattern with an identifier for the pattern. It groups the patterns into sets and stores them so that they can be used to populate the pattern space later. It also stores the list of identifiers in the order that they appeared. This will be used to form indexes in the database later. The identifiers for the patterns are simply integer values. However, they need to obey one simple rule which is that for two patterns p_1 and p_2 if

$$p_1 < p_2 \Rightarrow id(p_1) < id(p_2)$$

That is, the identifier scheme must reflect the natural ordering of the patterns. The identifiers do not need to be concurrent; in fact it can often be better if they are not.

When we run queries against a table in the Zizo database we do not reconstitute the original data to facilitate the query. To do so would restore the characteristics of a conventional relational database where results require time $O(n.\log(n))$. Instead we transform the query into an equivalent operation against the pattern space that can be run in time $O(n)$ or better.

We can see how this might work taking a very simple example. We will run the SQL query

```
SELECT * FROM table WHERE price>100
```

Conventionally we would have to visit each record, recover the price for that record, compare it to 100 and if it is greater add the record to the result set. This is how we go about it in the Zizo database. The first stage is to visit the pattern set for price and determine the identifier for the pattern 100 in the set (if the value isn't in the set then we just need any identifier that could be used for 100 if it were to appear in the set), call it *v*. Because the set is stored in ascending order of pattern we can do this in time $O(\log(p))$ where p is the number of patterns in the set. We also know that by definition $p \le n$, the number of rows in the table, then this operation is bounded by $O(\log(n))$. Now using the list of identifiers for the column price as an index we compute a list of integers which are the set of values such that

$$I' = \{i: 0 \le i \le n \wedge C(price)(i) > v\}$$

where *C(price)* is the list of identifiers for column price. The result from running the query on table <*I,M,P*> is simply the triple <*I',M,P*> so both the metadata and pattern space are reused.

© Zizo Software Ltd. 2015

This query was very simple, even trivial, but in the next section we shall look at how much more complex queries can run directly in the pattern space.

Partition

This algorithm is based on the original work by Wallace Feurzeig[1960] looking at the sorting of information in linear time. It has been enhanced in Zizo to create partitions and to deal with partial information see [2000] and [2004] for more details.

The partition algorithm takes a single Zizo table and maps it into a set of Zizo tables each of which has a unique characteristic. For example a partition by the field "city" will map the table into separate tables such that

$$parition(< I, M, P > ,\text{city}) \Rightarrow \{< I_k, M, P >: \forall r_1, r_2 \in < I_k, M, P > .\text{city}(r_1) = \text{city}(r_2)\}$$

And

$$\forall < I_j, M, P >, < I_k, M, P > \in partition(< I, M, P > ,\text{city}).\ \text{city}\big(< I_j, M, P >\big) = \text{city}(< I_k, M, P >) \Leftrightarrow j = k$$

Finally

$$\forall r \in < I, M, P >. \exists < I_k, M, P > \in parition(< I, M, P > ,\text{city}).\ r \in < I_k, M, P >$$

So the partition preserves all the records in the original and groups them according to the given field (city).

We will look at the following example. We start with a column and a pattern set associated with that column. Recall that for each pattern we also know the frequency of the pattern.

Column: Cities	Index: Cities	Partition
Liverpool	2	-
Birmingham	1	-
London	3	-
Manchester	4	-
London	3	-
Birmingham	1	-
London	3	-
Liverpool	2	-
Birmingham	1	-
Manchester	4	-
Manchester	4	-
Birmingham	1	-

Pattern Set: Cities

Pattern	Frequency
Birmingham	4
Liverpool	2
London	3
Manchester	3

Offsets
0
0
4
6
9

First generate a new empty index with 12 gaps.
And a table of offsets which is the running total of the frequencies (from null (pattern 0)).

© Zizo Software Ltd. 2015

We convert the frequencies into a set of offsets using the procedure

```
total=0
for (i=0...m)
  offsets[i] := total;
  total := total+freq[i];
```

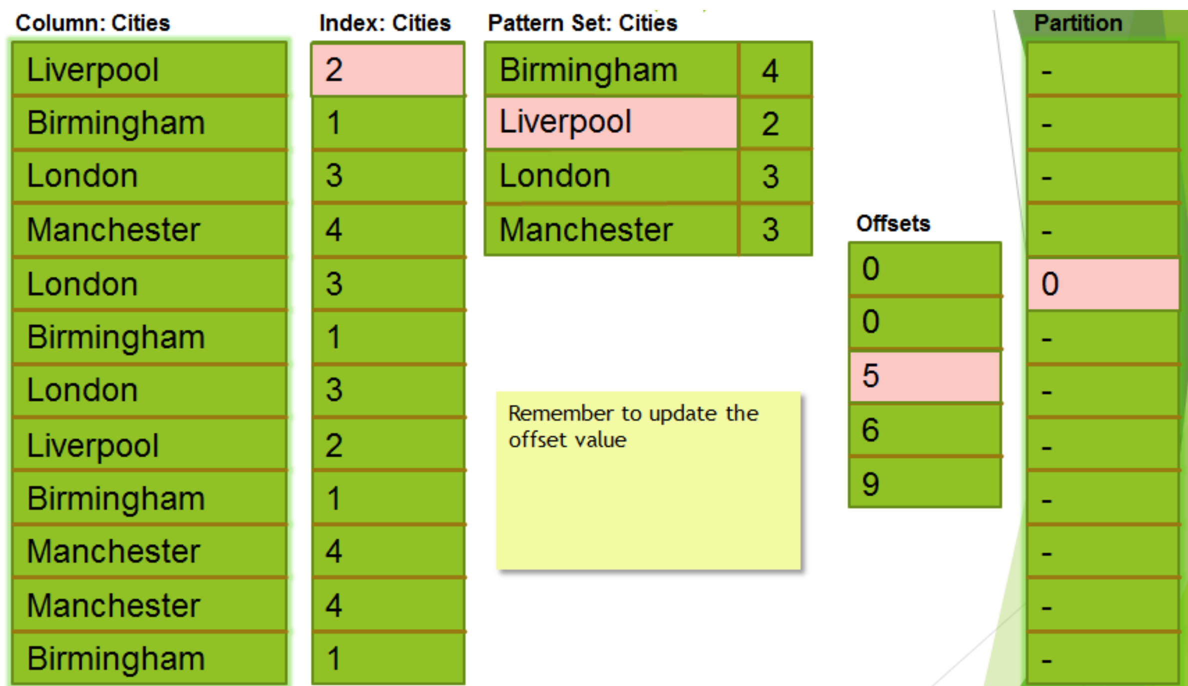
Note that the time to do this is $O(m)$ and $m \leq n$ so it is $O(n)$.

Now we partition using the following procedure

```
for (i=0...n)
  result[offsets[id[i]]++] = i;
```

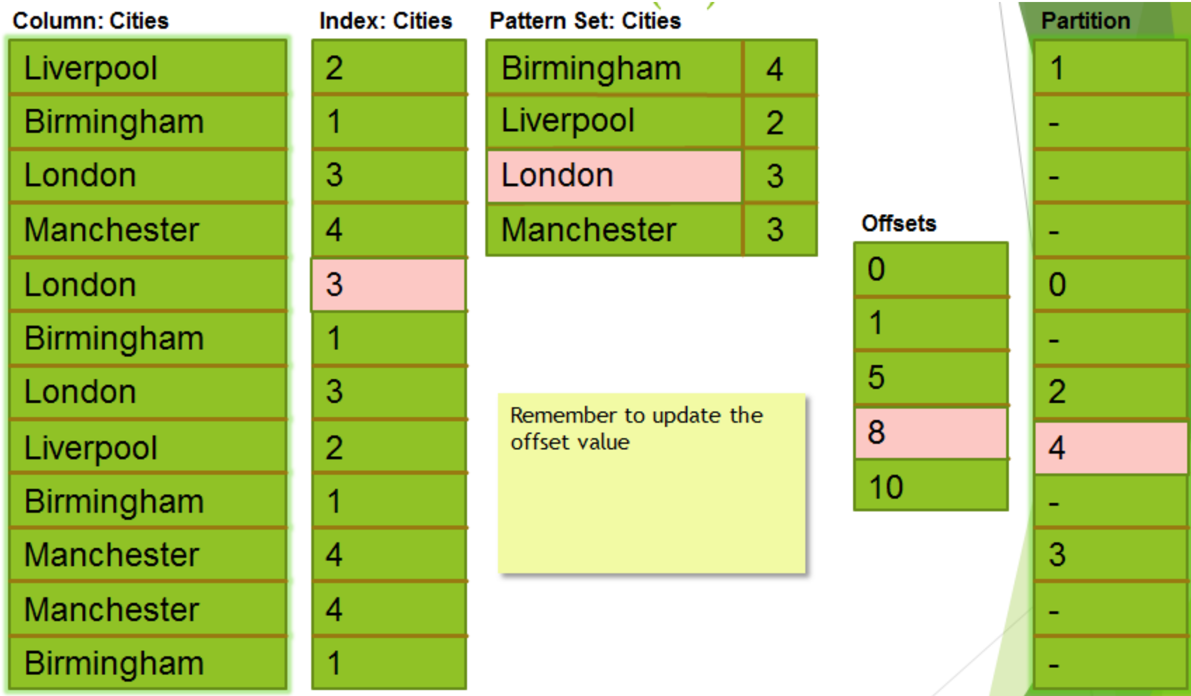
where id is a list of the pattern indices for the set (suitably compressed into a series that matches the offsets).

The attached figure shows the state of the partition after a single iteration of this loop



after 5 steps

© Zizo Software Ltd. 2015



and after all 12 steps

We have now performed a sort in linear time according to the method set out by Wallace Feurzig. However, to complete the partition we need to obtain separate tables for each sub-set. To do this we need to know the points in the sorted index where each subset starts and ends. Fortunately we find that the starts are all the positions in the original offsets array and the ends correspond with the values in the final offsets array. Therefore we can complete this partition by adding the sorted index into the pattern space and each partition is a table

$$\langle I_k, M, P+I' \rangle$$

where I' is the full index and I_k is the interval $[O_I[k], O_F[k])$ where O_i is the initial offsets and O_F is the final offsets.

© Zizo Software Ltd. 2015

Just being able to partition a dataset by a single field in linear time is good, but on its own it would not be sufficient to implement an entire RDBMS query engine. We need to examine some surprising properties of partition to see its true power. We will begin by listing out some of the common database operations that can be implemented solely using partitions.

- GROUP BY, obviously you can partition and then run the aggregates against each partition
- SORT, simply partition and use the master index as the representation of the sorted data set
- DISTINCT, partition and then form a new index taking just the first value from each partition
- INTERSECT, partition the two data sets and then use the classic merge algorithm to find the common algorithm. This is time O(n) to partition and also O(n) to do the merge, so it is O(n) overall
- DIFFERENCE, is essentially the same as intersect only treating the merge slightly differently
- JOIN, this is perhaps the most surprising result. To join two data sets A and B first partition A and B using the field (or fields) that form the join key. Now you form the join by processing the partitions from each table merging them. If two partitions have the same key, then add all the rows from the cross product of the partitions into the final join. The time for this operation is $O(n_A)+O(n_B)+O(n_J)$ where n_J is the size of the final join. This is not quite linear in the size of the initial data but if $n_J \geq n_A$ and $n_J \geq n_B$ then the time to do the join is $O(n_J)$ which is the theoretical limit on performance anyway and is linear in the size of final data. No other algorithm could improve on this.

All these results have depended on two so far unspoken assumptions which are that for any column we are partitioning by we have a properly annotated pattern set and we can find a partition still in linear time where it involves two or more columns. We will deal with the second assumption first. It turns out that for two or more columns we can compute a partition as a function of the partitions by the individual columns. For example

$$parition(< I, M, P >, [\mathrm{A,B}]) = parition(< parition(< I, M, P >, \mathrm{B}).index, M, P >, \mathrm{A})$$

(* this is analogous to the known result that $sort(T, [A, B]) = sort(sort(T, B), A))$

so partitioning in the reverse order of the columns we can still compute the function in time $O(n)$. During this operation it is rather more difficult to keep track of the locations of the boundaries between the partitions, but it can still be done in linear time (see [2012] for details).

As for the first assumption: we know that the build process provides us with patterns and frequencies for all the root pattern sets. However, if we were to attempt to partition a subset of the data we no longer have an accurate pattern set ab initio. It turns out that this is not a problem as long as for two tables

$$T' \supset T.\, f(a, T') \leq f(a, T)$$

where $f(a, T)$ gives you the frequency of pattern a in table T. Since T' is a subset of T, this is trivially true. Therefore to partition T' by column A we can actually partition it using the pattern set for A in T (which we already have). This will result in master index that contains gaps, but we can identify the gaps using the start and end values of the offsets, so we can partition any subset of the data in linear time too.

Our final hurdle is to deal with the situation where we are asked to partition by values which are not directly columns in the data but are computed as a function of one or more columns. The time to do this is dependent on the nature of the function. If f is a function of a single column and f is isotone (order preserving) then partition will still take time $O(n)$. If f is a function on a single column and anti-tone (order reversing) it is also linear. If it doesn't preserve order then the time is $O(m.\log(m))+O(n)$ where m is the size of the pattern set for the virtual column being created. Only in the case where we need to partition by a function of two or more columns are we forced into an operation with time $O(n.\log(n))$.

© Zizo Software Ltd. 2015

The Associative Store

So far we have shown how we can create a database where new patterns can be added into the database in time $O(n)$ (except in extreme circumstances). We also have an engine that runs queries by fetching patterns from the pattern space. The time to recover a pattern from the pattern space is $O(\log(p))$ where p is the size of the pattern space (based on the use of a self-balancing hashing algorithm). Now unless n is very small $\log(p)<n$ so it is always more efficient to recover a pattern from the pattern space than to directly query the table. The next thing to observe is that $O(\log(p))$ approximates the performance of typical NoSQL databases. Therefore Zizo can keep pace with a NoSQL database so long as the needed patterns are in the pattern space. Recall also that Zizo operates by decomposing queries into a series of patterns that must be either retrieved from the pattern space or computed from other patterns. While it is unlikely that for any given query every pattern it needs will be in the pattern space, we seek to ensure that as many of these patterns as are possible are present. In this way the performance of Zizo approaches $O(\log(p))$ but usually falls just short of it.

How do we make sure that the needed patterns are in the pattern space? First of all we store any pattern that we create in the pattern space. The rationale for this is that normal usage is to ask a question and then a follow-up question based on the previous results. Second we rely on the "wisdom of crowds". Having many users asking questions of the system, greatly improves the chances that one user has created patterns that another user can benefit from. Third where there are obvious derived patterns or optimisations that can be run in the background we will schedule these and add the results to the pattern space hopefully in time for one or more users to benefit from the new pattern(s).

Of course we can't keep adding patterns to the pattern space forever. This is where the extra annotations in the pattern space come into play. For every pattern we know the age of the pattern (time since it was last read), and the cost of creating the pattern. Based on that knowledge we can create a process that is searching for and removing the oldest and cheapest patterns from the pattern space. Thus we have one set of processes that are adding patterns to the pattern space based on what users are currently doing while another set of processes is removing patterns based on what users are not doing. In this way we strive to keep the pattern space as relevant as possible to the current query behaviour.

The net effect of this is that in use the query engine will find that 90% of the patterns it needs are already in the pattern space and that the remaining patterns can easily be created in a short time. Thus the query behaviour of Zizo approaches $O(\log(p))$. We could never achieve this limit unless we had infinite capacity and processing power, but with a large number of end users the average response time is close enough to the limit to be indistinguishable from it.

We will finish by looking at an example query that shows how simple things can be with a pattern database. We will look at the following query:

```
SELECT * FROM sales AS S WHERE
  cost > (SELECT AVG(cost) FROM sale WHERE sales.dept=S.dept);
```

This query will return all the rows in the sales table where the cost of that sale is higher than the average cost for a sale in that department. This is quite a difficult query to run efficiently in SQL, it must either be translated into a "self join" which is itself expensive to compute or the planner must be able to recognise the query and create a separate structure containing the average for each department which can be used to perform a series of selections against the table. However, with Zizo we don't need to do either of these. Instead we can simply run the query precisely as it was written. When we run of the first record (let us say that the department is 'A') then we must do

```
SELECT AVG(cost) FROM sales WHERE sales.dept='A';
```


© Zizo Software Ltd. 2015

then take the result and use it to choose whether or not to include the first row. Now for the second row (with department 'B') we will repeat running the inner select with department 'B'. Then for the third row which is for department 'A' again we must run

```
SELECT AVG(cost) FROM sales WHERE sales.dept='A';
```

again. However, now the result of this query is part of the pattern space, so Zizo will not actually run it a second time, but will simply read the result from the pattern space. In fact, the sub-query will be run just once for each department which is close to an optimal behaviour. This is all done without the need for complex planning and rewriting of the original query. The nature of Zizo is such that for most queries the pattern database naturally leads to good solutions without the need for anything but the simplest of planning.

Conclusions

The Zizo Database provides many of the performance benefits of a typical NoSQL database. When loaded and with sufficient memory and users, the user will experience performance that scales $O(\log(n))$. However unlike most NoSQL databases Zizo will run SQL and store structured data.

As well as supporting SQL querying, Zizo's incremental way of building up queries allows for other approaches to data discovery where the final question is shaped by the questions that have gone before.

The absence of fixed schemas in the Zizo database along with "no indexes" means that the database is low maintenance. There is no data design required when loading data nor is there any need to optimise for performance.

Finally by supporting SQL as a way to query Zizo we open it up to a wide range of third party tools from visual query designers to component frameworks and business intelligence tools. The way Zizo works means that without making modifications to these tools they can benefit massively from Zizo's performance.

References

[1960] Wallace Feurzeig, “Math Sort”, Communications of the ACM, Volume 3 Issue 11, Nov. 1960, Page 601

[2000] Tom Longshaw, “Method of querying a structure of compressed data”, International Patent, WO 02/063498 A1 priority 2001

[2004] Tom Longshaw et al, “Method and system for implementing an enhanced database”, US Patent 60/671,172, priority 2004

[2012] Tom Longshaw, “A method of querying a data structure”, European Patent EP 2 618 275, priority 2012

© Zizo Software Ltd. 2015