Performance Tuning an Algorithm for Compressing Relational Tables

Authors

Jyrki Katajainen and Jeppe Nejsum Madsen

Speaker

Jeppe Nejsum Madsen

Relations

A relation consists of a scheme and an instance:

- A scheme is a finite set of attributes. Each attribute is associated with a set of values, called its domain.
- A tuple over a scheme is a mapping that associates with each attribute of the scheme a value from the corresponding domain.
- An instance over a scheme is a finite set of tuples over that scheme.

Relation Optimization Problem

Input: A relation R

Output: A compressed representation of R that supports the relational operations needed on the data. In our case σ (Select), π (Project) and $\bowtie$ (Join).

Our Motivation

Our motivation is constraint satisfaction problems (CSPs), where relations are used to store valid variable assignments.

- Large solution space: An unconstrained CSP with n boolean variables has 2^n possible solutions.
- Larger problem instances can be handled by compressing relations.

Compression Using Cartesian Products

Idea: Use Cartesian products to generate the set of tuples.

Example: Given a relation with tuples

$$\{(0,0,0),(0,0,1),(0,1,1),(1,0,1),(1,1,1)\}$$

we can generate the tuples using Cartesian products:

$$\{(0,0,0)\} \cup (\{0,1\} \times \{0,1\} \times \{1\})$$

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

A	B	C
0	0	0
$\{0,1\}$	$\{0,1\}$	$\{1\}$

Our Contribution

- A detailed analysis of an algorithm that implements a compression heuristic described by [Møller 1995].
- Propose a new algorithm that improves the running time of the original algorithm while, with high probability, producing the same output.
- Provide an implementation of our algorithm in C++.
- Compare the running times of our implementation with existing implementations, one which is used in a commercial software product. Significant speedups can be observed on all data sets used.

The Compression Heuristic

The heuristic works by compressing each column in turn. The work falls in two phases:

Phase 1: The relation is analyzed to determine the order in which the columns are to be compressed.

Phase 2: The relation is compressed on each column according to the selected column order.

Phase 1

In phase 1 we determine the number of unique tuples in each attribute's complement.

The complement of a relation R with respect to an attribute A is the tuples of R with the values corresponding to A removed.

A	B	C
0	0	1
0	1	1
1	1	1

Complement wrt. B

A	C
0	1
0	1
1	1

In the example above there are 2 unique tuples in B's complement.

Phase 2

In phase 2 the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements.

Phase 2

In phase 2 the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements.

Consider column B:

A	B	C
0	0	1
0	1	1
1	1	1

Phase 2

In phase 2 the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements.

Consider column *B*:

A	*B*	*C*
0	0	1
0	1	1
1	1	1

Unique tuples in *B*'s complement:

A	*C*
0	1
1	1

Phase 2

In phase 2 the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements.

Consider column B:

A	B	C
0	0	1
0	1	1
1	1	1

Unique tuples in B's complement:

A	C
0	1
1	1

Construct Cartesian Product:

A	B	C
0	$\{0, 1\}$	1
1	$\{1\}$	1

Phase 2

In phase 2 the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements.

Consider column *B*:

A	*B*	*C*
0	0	1
0	1	1
1	1	1

Unique tuples in *B*'s complement:

A	*C*
0	1
1	1

Construct Cartesian Product:

A	*B*	*C*
0	{0, 1}	1
1	{1}	1

Analysis: Phase 1

For the purpose of analysis, we assume that the input relation is uncompressed and does not contain any identical tuples. Let k denote the number of attributes and n the number of tuples in the relation.

Two methods:

- Using Vector sorting. Worst case running time $O(k^2 n + kn \log_2 n)$.
- Using a dictionary. $O(k^2 n)$ expected running time, $O(k^2 n \log_2 n)$ worst case.

Analysis: Phase 2

For each column, we maintain a dictionary with the complement tuples as key.

- The number of scalar values is bounded by kn.
- Keep sets sorted: Comparison is linear in the size of the searched tuple.
- Sorting cost is $O(n \log_2 \min\{d_{\max}, n\})$, where $d_{\max}$ is the size of the largest domain.
- Lookups and possible inserts for all tuples take $O(kn)$ expected time.

Total running time of Phase 2 is $O(k^2 n + kn \log_2 \min\{d_{\max}, n\})$ in the average case, $O(k^2 n \log_2 n)$ in the worst case.

Improving Phase 1

Idea: Compute an approximation to the number of unique tuples in the complement.

Method:

- Use a hash function to compute a signature for each tuple in the complement.
- Use the number of unique signatures as an approximation for the number of unique tuples.

Strongly Universal Hashing

[Carter & Wegman 1981] Let U and T be subsets of the natural numbers. A class $\mathcal{H}$ of hash functions from U to T is said to be strongly universal if a randomly chosen hash function h from $\mathcal{H}$ maps elements pairwise independently, i.e., for all $x, y \in U$, $x \neq y$, and for all $\alpha, \beta \in T$:
$\Pr(h(x) = \alpha \text{ and } h(y) = \beta) = O(1/|T|^2)$.

Supports vector hashing [Carter & Wegman 1979]: Let $\mathcal{H}^q$ denote the class of hash functions from U^q to T such that $(h_1, \ldots, h_q)(x_1, \ldots, x_q) = h_1(x_1) \oplus \cdots \oplus h_q(x_q)$ where $\oplus$ is the binary XOR operation and $h_i \in \mathcal{H}$ for all $i \in \{1, \ldots, q\}$. If $\mathcal{H}$ is strongly universal, then $\mathcal{H}^q$ is strongly universal.

Computing the Signatures

Notation:

$h_*(r_{i*})$: Vector hash value for the ith tuple

$h_j(r_{ij})$: Hash value for the ith tuple and the jth attribute using a strongly universal hash function h_j

$h_{\bar{j}}(r_{i*})$: Signature for the ith tuple in the complement with respect to the jth attribute

We then have

$$\begin{aligned} h_*(r_{i*}) &= h_1(r_{i1}) \oplus \cdots \oplus h_k(r_{ik}) \\ h_{\bar{j}}(r_{i*}) &= h_1(r_{i1}) \oplus \cdots \oplus h_{j-1}(r_{i,j-1}) \oplus h_{j+1}(r_{i,j+1}) \oplus \cdots \oplus h_k(r_{ik}) \\ &= h_*(r_{i*}) \oplus h_j(r_{ij}). \end{aligned}$$

The signatures for all k complements are computed in $O(kn)$ time in the worst case.

Improved Phase 1

Proposition: For a signature universe T, for which $|T| = n^{2+\varepsilon}$ for $\varepsilon > 0$, the probability that the outcome of our modification is the same as that of Phase 1 of Møller's heuristic is at least $1 - 1/n^{\varepsilon}$. The worst-case running time of our modification is $O((2+\varepsilon)kn)$.

Algorithm Engineering

The algorithm has been implemented in C++ using various tricks in order to speed execution:

- Template meta programming is used to inline and specialize inner loops.
- For attributes with small domains, sets are stored using fixed size bit vectors.
- Hash functions are tabulated and only table lookups and XORs are needed to compute the hash value.

Full details can be found in the technical report available at `www.cphstl.dk`.

Performance Study

Characteristics of the input data.

Instance	k	$\sum_{i=1}^{k} d_i$	n	n_c	n_c %
heq	10	1643	151374	5020	3.3%
plan31	14	28	8192	14	0.2%
q10a	8	80	149552	13144	8.8%
q10b	8	80	55658	6632	11.9%
ns11	11	65	333322	102	0.03%

Speedup factors. Current is used as base.

Instance	APL	Current	Tuned	Approx
heq	0.33	1	4.6	13.2
plan31	0.11	1	9.4	17.1
q10a	0.32	1	42.5	77.1
q10b	0.21	1	15.7	22.9
ns11	0.57	1	65.1	196