A Novel Method for Calculating Query Hashes for Improved Query Grouping in Relational Database Management Systems

Derek Colley M.Sc.
Department of Computing, School of Digital, Technologies and Arts,
Staffordshire University, College Rd., Stoke-on-Trent, ST4 2DE, U.K.
derek.colley@staffs.ac.uk

Md Asaduzzaman Ph.D.
Department of Engineering, School of Digital, Technologies and Arts,
Staffordshire University, College Rd., Stoke-on-Trent, ST4 2DE, U.K.
md.asaduzzaman@staffs.ac.uk

Abstract

Database queries are stored and compared by relational database management systems as hashes, or short unique representations, of the original query text. This leads to cache misses and increased resource consumption by database engines when queries differing only in non-syntactic detail, or queries which are relationally equivalent, are presented to the query parser. We propose a new method of structural query decomposition, transforming database queries into multidimensional adjacency cubes (MACs), allowing the codification of queries by structure rather than content as currently implemented. We build and test our solution, demonstrating superior query hash grouping to that currently offered by a leading relational database platform, and consider the applications of this new technique.

Keywords

Database query representation, query hash generation, query similarity, database query grouping, query caching.

Introduction

In Relational Database Management Systems (RDBMSs), the plan cache is used to store hashed database queries and accompanying execution plans [1] for the purposes of faster query execution performance when recompilations of the queries are avoided. Combined with forced parameterization, this encourages the maximum amount of query plan cache re-use given any incoming query workload.

However, despite database queries being extrusions of the relational algebra [2], albeit often with language-specific additions or amendments, a computational approach is not used when queries are compared to existing queries in the cache. Instead, queries are treated as if they were natural language constructs. This leads to inefficiencies; differences in column selection, join construction or even whitespace within two queries leads to the misidentification of these queries as separate constructs and plan recompilation can be forced as a result, worsening query performance. When parameterisation is absent, this effect is amplified [1]. This issue is exacerbated by object-relational mapping frameworks which produce database queries automatically at runtime, often using anti-patterns such as nesting or row-by-row query production to the detriment of query plan re-use [3, 4, 5].

In this paper, we present a novel method of query representation using a three-dimensional adjacency matrix to represent structural similarity between objects within queries, intended as an alternative to the existing hashing and comparison process within RDBMSs. By using concepts from the relational algebra [2], we show how similar queries can be grouped by examining the relationships between objects within the query, in contrast to text comparison, and consequently query plan re-use can be significantly improved. We test our approach against Microsoft SQL Server 2017.

Related Work

Relational database queries are structured blocks of text that follow a set of syntactic rules [6], although RDBMSs diverge from the SQL standard in many key areas, particularly syntactically. When a query is presented to an RDBMS from a calling application, it is first treated as narrative text and parsed, then bound. The algebriser creates a 'query fingerprint' – a hash – and checks the plan cache for the existence of the hash [1]. Should the hash already exist, the next steps in the query execution cycle are skipped and the execution plan already associated with the found hash is re-used.

This tokenisation, algebrisation and hashing of the query is not seamless. Minor variations in query presentation, unrelated to the relational form of the query, can cause cache misses. The change in use from static, inline queries to dynamically generated queries from ORM frameworks together with increases in velocity, volume and variety [3, 7] increase the heterogeneity of queries presented to the relational database layer and as such contribute to the deficit in query plan re-use.

The general technique for semantic parsing was introduced by DeRemer [8] from a seminal paper by Knuth [9] on general LR-type parsers. The 'LALR(1)' parser is a simplified left-to-right, bottom-to-top parser of a token stream that does not require backtracking to apply rules and is memory-efficient. In relational databases, techniques including Yacc compilers [10] are used to generate parse trees. The resultant trees are stored in a parsing table for use by the next stage of the

query optimisation process. With minor variations in the entry points for the parser and the resulting data structures, this parsing process is identical across various database management systems.

Repeated presentations of a query can result in repeated parsing and optimisation despite the process having already been followed for antecedent queries which are structurally, semantically, or functionally identical. The use of a better query comparison method, one based on computational query representation rather than semantic query representation, has the potential to reduce the proportion of query recompilations. The ability to identify similar queries yields advantages such as the re-use of a previously generated execution plan, eliminating the optimisation steps and lowering the time taken to process the query, and the ability to cache the intermediary objects such as the parse tree which reduces the space required for the plan metadata in memory, increasing memory capacity for other queries.

In some implementations, queries can also be prepared, or parameterised. The process of preparing queries means to identify the parameters within the query and remove them to a separate list of key-value parameter pairs, to be substituted into the query at run-time. There are advantages to this approach including greater query re-use and interoperability with wider data processing platforms [11]. However, this approach is dependent on the implementation of the RDBMS. An additional mitigation is that queries which are frequent often refer to objects which have their pages stored in a buffer cache meaning a large portion of data retrieval can take place in-memory without reference to the disk subsystem, significantly reducing I/O costs and reducing the query execution time to the advantage of the user.

Checks for query similarity are primitive by design because the query execution process by necessity must be extremely swift and so an excessive level of query pre-processing would impact overall query execution time. The limitations of current approaches in query parsing, storage and recompilation open new avenues for exploring query parsing alternatives and supplementary techniques for reducing the workload sent to the query parsing process through query pre-processing.

Methodology

We propose an alternative method of relational query representation, based on a) identifying the relationships between elements in the query and b) describing the type of relationship, the whole to form a directed graph. In such a representation, each object in the query (column name, or table or view name) becomes a node in the graph, and the relationship type between nodes is categorised as either:

- *Membership* (column name is a member of a table)
- *Intersection* (a relationship between two tables, typically an inner or outer JOIN)
- *Predication* (a condition, by way of an operator such as =, < or >, is placed on the relationship)
- *Projection* (the node is a subset of another node).

Although this structure resembles a parse tree (an acyclic graph), it is constructed from the objects and the type of relationship they have with each other, rather than the relational operators alone, and consequently has a different abstract (and internal) representation.

This directed graph can be represented in terms of the adjacency of the nodes, in a construct called an adjacency matrix. It can be represented as a 3-dimensional binary adjacency matrix (termed a "multidimensional adjacency cube", or MAC), which represents the structure of the query in a binary medium. The *i* and *j* axes are comprised of an ordered node list, and the *k* axis is a type-representation. We notate |Cx| to mean the node cardinality, or number of nodes, of any given cube Cx. Thus, any intersection of the three axes *i*, *j* and *k* indicates a relationship exists and contains the value 1, and all other intersections contain 0. The result is a multidimensional list. The process is illustrated in Fig. 1.

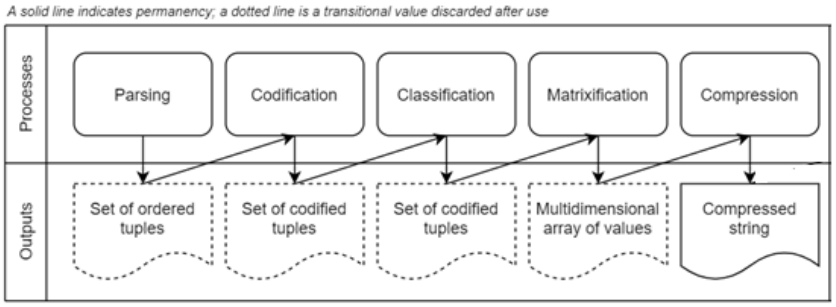

Fig. 1: Process flow of the query transformation process

Our process is split into five distinct steps.

Parsing: Using linear tokenisation, we split the query into distinct atomic elements, or *words*. We state that this parsing process *P* takes as input a query *Q* which consists of a set of words *w*. We apply a series of functions *f* over combinations of *w* in *Q* to produce a set *S* of tuples *t*, of which each *t* consists of exactly three values t_1, t_2 and t_3 – two objects, and a type description derived from the purpose of the object pair in the query (1).

$$P = \forall f(w \in Q) \rightarrow S \text{ and}$$
$$S = \forall t \in S,\ t = (t_1, t_2, t_3) \qquad (1)$$

Codification: We codify each object and each operator with a shorthand notation. We replace literals with a single non-unique shorthand placeholder. We state that for all object members (t_1, t_2) of set S, we replace each t_1, t_2 with a codification of t_1, t_2, designated $c(t_1)$ or $c(t_2)$. t_3 is left intact (2):

$$S = \forall t_1, t_2 \in S, t_1 = c(t_1), t_2 = c(t_2),\ t_3 = t_3 \qquad (2)$$

Classification: For each tuple in S, we classify each object o_1 in each tuple as either a selection, a member, a predicate or an intersection on object o_2 in the tuple. Each of these terms are used in their relational or set-theoretic sense; a selection is $\boldsymbol{\sigma}$ of values over a relation R; a member is an element x that belongs to a set A such that x ∈ A; a predicate is a condition placed on a selection or more formally, the expression that is φ in the selection $\boldsymbol{\sigma}$ of values over a relation R subject to the propositional expression φ; and an intersection is a natural join ⋈, theta join θ, semi-join ⋉ and ⋊, left-outer and right-outer join ⟕ and ⟖ (we exclude the anti-join ▷). The output is a temporary set, which we then order alphabetically by object code and deduplicate. This set, designated *K*, consists of a distinct list of object pairs (o_1, o_2) and a classification *c* arranged as a tuple (3).

$$\forall \mathrm{o} \in \mathrm{S},\ K = f(x)f(y)\{o^1(o_1, o_2, c),\ \ldots\ o^n(o_1, o_2, c)\}$$

[where c ∈ C ('selection','membership', 'predication','intersection') (3)
and f(x) f(y) represent deduplication and ordering, respectively]

Matrixification: The matrixification function *f(K)* arranges each object on virtual X, Y and Z axes with every object appearing on both X and Y axes in every Z slice. The Z axis has a cardinality |Z| of 4, consisting of a slice for each classification, selection, membership, predication and intersection. For every relationship on axes X, Y and Z, we mark the value at the intersection with the value 1. We mark all other intersections with the value 0. The output is a 3-dimensional matrix M which we represent as two matrices below, showing axes XY and YZ (4).

$M = f(K)$, such that:

$M(x) =$ ordered set of $\forall o \in K$ and

$M(y) = M(x)$ and

$|M(x)| = |K| \therefore |M(y)| = |K|$ and

$|M(z)| = 4$

such that the values in M consist of (XY, YZ):

$$M = \underbrace{\begin{pmatrix} [0 \vee 1] & \cdots & [0 \vee 1] \\ \vdots & \ddots & \vdots \\ [0 \vee 1] & \cdots & [0 \vee 1] \end{pmatrix}}_{(XY)} \underbrace{\begin{pmatrix} [0 \vee 1] & \cdots & [0 \vee 1] \\ \vdots & \ddots & \vdots \\ [0 \vee 1] & \cdots & [0 \vee 1] \end{pmatrix}}_{(YZ)} \quad (4)$$

Compression: We combine the ordered matrix of objects (in shorthand notation) in a string format and the resulting binary expression, read left-to-right (X), top-to-bottom (Y), front-to-back (Z) as a hexadecimal value.

$$S' = \forall \, \mathrm{m} \in M, hex(concat(m)) \quad (5)$$

Implementation

We began by creating a sample schema based on the entities of a Sale, a Product, and a Customer. Sale is split into a strong entity and a weak entity, reflecting that SaleLineItem is a hierarchical child of Sale, the latter we term SaleHeader.

Our experiments are carried out using the Transact-SQL syntax on Microsoft SQL Server 2017 Developer Edition, with our algorithms coded in Python 3.8.2. We wrote an implementation of the MAC generator which ingests a single SQL query and outputs a hexadecimal query hash calculated from a multidimensional list object containing the binary intersections of each tuple, as described in our methodology.

To test the process, we generated 10,000 valid queries against our sample schema. Our generator used the schema definition and created queries subject to the following limitations, which match the limitations of our cube generator implementation: up to 2 joins, 12 predicates (including primitives and Booleans), 10 columns in the selection, excluding subqueries, CTEs, aggregates and side-effecting functions (see Future Work). Literals were randomly generated. We then ran timing tests (determining MAC execution speed); cache tests (determining existing database optimiser efficiency in grouping queries); MAC generation

tests, determining the efficiency of our process in grouping queries) and pair sampling tests (determining the validity of our process in grouping relationally-identical queries together). Our results follow.

Results

Timing test We iterated through each of the 10,000 queries in the pool using the MAC generator, capturing MAC process execution duration per query. We switched off console output to ensure parity with the database engine, where no console output is generated on query parsing.

We found that 10,000 executions were successful (100.0%). Of these, 48.86% completed in under 0.5ms; 71.19% completed in under 1.0ms; 91.53% completed in under 1.5ms; 95.13% completed in under 2.0ms with all outliers taking no longer than 4.0ms to complete, comprising the remaining 4.87% of the total. The mean average time to complete was 0.6ms (median 0.9ms, which may be more accurate given a standard deviation of 0.65ms, or 1.1 standard deviations from the mean, with variance of 0.42ms). Our results are shown in Fig. 2.

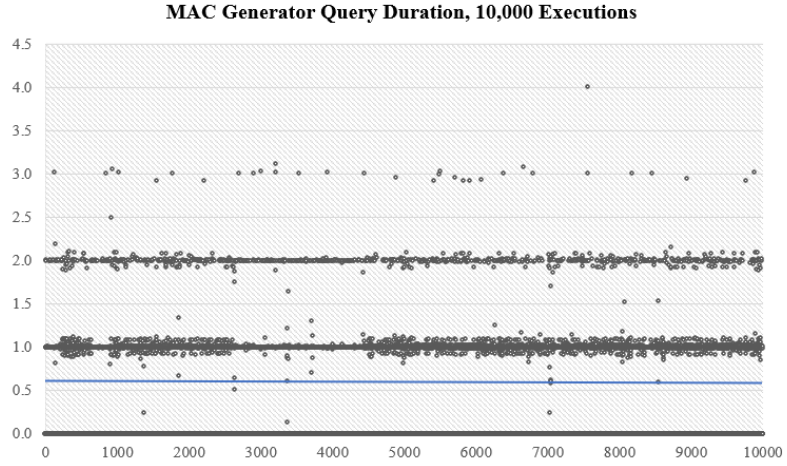

Fig. 2: Results from timing test

Cache test We next selected 1,000 random queries from the pool and executed them. We used a subset since Microsoft SQL Server ages out queries from the cache rapidly, depending on a proprietary algorithm; through trial-and-error

we have found a range of 1,000-1,500 to fit comfortably within the cache, following a cache flush, before removal takes place (given no other load).

We measured the number of non-distinct query representations retained in the cache. We were able to retrieve 1,000 queries (100%) from the cache as expected.

We found every query had a distinct SQL handle, plan handle and query hash. This meant that during execution, even with forced parameterisation switched on, SQL Server was *unable to group or classify queries which, relationally, are identical* (different column arrangements; different predicate orders; different literals; inclusion of whitespace and/or carriage returns). For the purposes of parsing and recompilation, each query had an independent hash generated and there were no instances where a query handle or query hash were shared, or duplicated, by any given SQL query. Every query hash was distinct and the efficiency of the database query engine at determining like queries was therefore zero.

MAC generation test We next tested the efficiency of our new process to determine whether it was able to classify and group queries using the structural approach. We took a different random subset of 1,000 queries from the query pool and iterated through them, storing the generated hash of each query (the SQL Server equivalent of the query hash). We repeated our analysis of the hashes to determine if any grouping took place. We found the MAC classifier was able to group 1,000 queries into 112 distinct groups based on the structure of the query.

Paired sampling test We examined the grouped queries and sampled pairs from each group. Each pair of queries resulted in the same MAC hash. We examined every hash where at least two queries had the hash – a total of 80 hashes, from 112 (71.4%). For each of these hashes, we selected two queries at random and compared them manually to determine if they were truly relationally identical or whether our process had miscategorised them, and if so, in what manner. We classified the differences into 9 categories as shown in Table 1.

We found 27 pairs of 80 (33.6%) of queries that the MAC process grouped were both relationally and structurally identical; selection ordering, predicate ordering and predicate literals are disregarded for set equivalence as per Codd's model of relational algebra [2]. Of the remainder, 10 of 80 (12.5%) had hash collisions, resulting in set inequivalence; a further 35 (43.8%) differed on join direction; with the remainder having identical hashes for different predicates, also resulting in structural difference, or set inequivalence. The overall success rate for the MAC generator in correctly grouping queries was therefore 33.6%, in comparison to Microsoft SQL Server, which was unable to group queries at all (0.0%).

Table 1: Results from paired sampling tests

Differences observed	Count of pairs	Result
Joins	2	Structurally different
Predicates*	3	Structurally different
Column ordering, joins	12	Structurally different
Joins, predicates	5	Structurally different
Chance collision (different relation)	10	Structurally different
Column ordering, joins, predicates	21	Structurally different
Column ordering	13	Structurally identical
Column ordering, predicates	13	Structurally identical
Identical, except predicates or literals	1	Structurally identical
Total query pairs	**80**	
Queries structurally different	53	
Queries structurally identical	27	

** For predicates to differ, each member of a pair must have different columns involved in the predicate(s); column ordering, primitive operators and literals are disregarded as they are abstracted in both the relational algebra and query execution plans and therefore these queries are regarded as structurally identical.*

Conclusions and Future Work

In answer to the increasing variety and volume of queries presented to the relational database perimeter, we proposed a novel method of query deconstruction using structure rather than semantic content. By graphing query structure as a multidimensional adjacency cube (MAC), we show how queries that differ only in predicate ordering, trivial construction differences such as the presence of arbitrary whitespace, and which differ in literals can be grouped and hashed.

We built and demonstrated the MAC generator against a realistic 3NF schema. We wrote and executed 10,000 queries, noting the query and plan representations in Microsoft SQL Server. For a subset, we noted how each query had a distinct representation and grouping was only successful at the plan hash stage. We noted the RDBMS was unable to group any SQL query hashes, whereas our approach demonstrated grouping of 1,000 queries into 112 groups with a reasonable degree of overall success (33.7%) in correctly classifying like queries.

This ability to group queries structurally has an abundance of potential applications. Future work in this area includes development of the MAC parser to solve the join differentiation and small MAC group collisions observed; extending design to subsets, aggregates and the full SQL syntax; including the full range of relational algebra and applying this technique to query performance optimisation,

for example the ability to compare queries using this technique to automatically select appropriate sub-schemas or alternative data partitions based on the performance of previously-run queries that are structurally similar.

References

[1] Delaney, K., Beauchemin, B., Cunningham, C., Kehayias, J., Nevarez, B. and Randal, P., 2013. Microsoft SQL Server 2012 Internals, O'Reilly Media, pp.703-715.

[2] Codd, E.F., 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, vol. 13, issue 6, pp. 377-387.

[3] Ireland, C., Bowers, D., Newton, M. and Waugh, K., 2009. A classification of object-relational impedance mismatch. *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, pp. 36-43.

[4] Chen, T.H., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M. and Flora, P., 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. *Proceedings of the 36th International Conference on Software Engineering*, pp. 1001-1012.

[5] Colley, D., Stanier, C. and Asaduzzaman, M., 2020. Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks, *Journal of Database Management*, 31(4), pp.1-23.

[6] International Standards Organisation (ISO), 2011. ISO/IEC 9075-1: 2011: Information Technology -- Database Languages -- SQL -- Part 1: Framework (SQL/Framework) (JTC1/SC32). [Online]. Available at: https://standards.iso.org/ittf/PubliclyAvailableStandards/c053681_ISO_IEC_9075-1_2011.zip [Accessed 06 Feb. 2021].

[7] Khan, M., Uddin, M., and Gupta N., 2014. Seven V's of Big Data; Understanding Big Data to extract value. *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education.* [Online]. Available at: [Accessed 18 Feb. 2021].

[8] DeRemer, F.L., 1969. Practical translators for LR (k) languages (Doctoral dissertation, Massachusetts Institute of Technology). [Online]. Available at: https://core.ac.uk/download/pdf/81140495.pdf [Accessed 06 Feb. 2021].

[9] Knuth, D.E., 1971. Top-down syntax analysis. Acta Informatica, vol. 1, issue 2, pp.79-110. [Online]. Available at: http://dcc.ufrj.br/~fabiom/comp20122/knuth_topdown.pdf [Accessed 10 Feb. 2021].

[10] Johnson, S.C., 1975. Yacc: Yet another compiler-compiler. Bell Laboratories. [Online]. Available at: http://web.wlu.ca/science/physcomp/ikotsireas/CP465/W3-BNF-LEX-YACC/Yacc_Introduction.pdf [Accessed 13 Feb. 2021].

[11] Meijer, E., Beckman, B. and Bierman, G., 2006. Linq: reconciling object, relations and XML in the .NET framework. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 706-706.