Efficient and Portable Einstein Summation in SQL

Mark Blacher
University of Jena
mark.blacher@uni-jena.de

Joachim Giesen
University of Jena
joachim.giesen@uni-jena.de

Julien Klaus
University of Jena
julien.klaus@uni-jena.de

Christoph Staudt
University of Jena
christoph.staudt@uni-jena.de

Sören Laue
Technical University of Kaiserslautern
laue@cs.uni-kl.de

Viktor Leis
Technical University of Munich
leis@in.tum.de

ABSTRACT

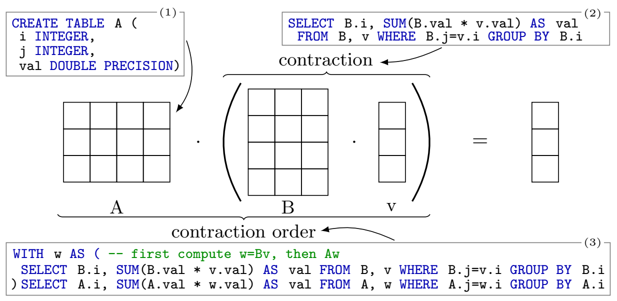
Computational problems ranging from artificial intelligence to physics require efficient computations of large tensor expressions. These tensor expressions can often be represented in Einstein notation. To evaluate tensor expressions in Einstein notation, that is, for the actual Einstein summation, usually external libraries are used. Surprisingly, Einstein summation operations on tensors fit well with fundamental SQL constructs. We show that by applying only four mapping rules and a simple decomposition scheme using common table expressions, large tensor expressions in Einstein notation can be translated to portable and efficient SQL code. The ability to execute large Einstein summation queries opens up new possibilities to process data within SQL. We demonstrate the power of Einstein summation queries on four use cases, namely querying triplestore data, solving Boolean satisfiability problems, performing inference in graphical models, and simulating quantum circuits. The performance of Einstein summation queries, however, depends on the query engine implemented in the database system. Therefore, supporting efficient Einstein summation computations in database systems presents new research challenges for the design and implementation of query engines.

ACM Reference Format:
Mark Blacher, Joachim Giesen, Julien Klaus, Christoph Staudt, Sören Laue, and Viktor Leis. 2025. Efficient and Portable Einstein Summation in SQL. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Relational databases are the backbone of our data-driven world and its data-intensive applications. SQL is the common language used in relational databases to store, manipulate and retrieve data. SQL is also suitable for simple computations with data. However, compared to procedural programming languages, SQL's data processing capabilities are limited. Therefore, practitioners and researchers strive to find SQL-friendly algorithms [1, 2], code-mapping approaches [3, 4], or even extend SQL's data processing capabilities by introducing new SQL operators and data types [5, 6].

Conference'17, July 2017, Washington, DC, USA
© 2025 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



```
CREATE TABLE A (
 i INTEGER,
 j INTEGER,
 val DOUBLE PRECISION)
```

```
SELECT B.i, SUM(B.val * v.val) AS val
 FROM B, v WHERE B.j=v.i GROUP BY B.i
```

```
WITH w AS ( -- first compute w=Bv, then Aw
 SELECT B.i, SUM(B.val * v.val) AS val FROM B, v WHERE B.j=v.i GROUP BY B.i
) SELECT A.i, SUM(A.val * w.val) AS val FROM A, w WHERE A.j=w.i GROUP BY A.i
```

Figure 1: Correspondence between fundamental SQL constructs and Einstein summation: (1) tensors are realized as relations, (2) tensor contractions are implemented as inner joins paired with GROUP BY clauses and the SUM function, and (3) common table expressions are used to optimize the tensor contraction order.

Bringing computation to the data has many advantages, especially when computations can be expressed in pure and thus portable SQL code. Processing data exclusively with SQL, avoids costly data transfer to external applications, data duplication, and possible data breaches. SQL provides a high level of abstraction from the underlying hardware. Vectorized, parallel or even distributed execution of SQL is done automatically by the underlying query engine. SQL computations can achieve excellent performance with powerful query engines [3]. Unlike user-defined functions, computations performed in pure SQL require minimal user permissions. Furthermore, pure SQL code avoids expensive context switches between the SQL parts and the procedural parts of the code, as they occur in user-defined functions [7].

Many of today's problems require efficient computations with multidimensional arrays, that is, tensors. In artificial intelligence and quantum physics, Einstein notation is commonly used to express large tensor expressions. Einstein notation is powerful enough to represent all kinds of multiplicative tensor computations in a natural and unambiguous way (see Background section). Although Einstein notation has been around for over 100 years, it did not reach the mainstream of the data science community until 2011 with its introduction in NumPy [8]. Since then, the notation has become increasingly popular and is now part of major machine learning and artificial intelligence frameworks [9–11], as well as numerous array computing libraries [12, 13]. However, these frameworks and libraries do not support Einstein notation with sparse tensors. Surprisingly, SQL is sufficient to evaluate arbitrarily complicated tensor expressions in Einstein notation, both for sparse and dense tensors.

In this paper, we present a code-mapping approach for Einstein notation that enables efficient Einstein summation in SQL. We use the term Einstein summation to refer to the actual evaluation of a tensor expression in Einstein notation. Mapping Einstein notation to SQL allows databases to perform the tensor computations without the need for external libraries. We use only portable SQL constructs for the mapping, so that the generated tensor computations can be executed on different DBMSes without any code adjustments. For implementing Einstein summation in SQL we exploit a natural correspondence between fundamental SQL constructs and Einstein summation. As illustrated in Figure 1, the correspondence works on three levels: (1) relations correspond to tensors, (2) inner joins paired with GROUP BY clauses and the SUM function correspond to tensor contractions, and (3) common table expressions correspond to optimizing the tensor contraction order, that is, enforcing query optimizations externally in queries. We explain how this correspondence can be exploited for efficient mappings from Einstein summation to SQL in Section 3. Note, however, that Einstein summation is not only used for small linear algebra expressions with vectors and matrices, as is the case in Figure 1. Instead, practical Einstein summation problems often consist of expressions with hundreds or even thousands of higher order tensors. Therefore, using an efficient contraction order to evaluate an Einstein summation query is crucial for its performance.

Executing Einstein summation in databases opens up new application areas for SQL-only computations. Einstein summation can be used for stand-alone computations or as part of larger algorithmic computations. In the Experiments section, we apply stand-alone Einstein summation in SQL to four recently published use cases, namely querying triplestore data [14], solving Boolean satisfiability problems (SAT) [15], performing inference in graphical models [16], and simulating quantum circuits [17]. The source code for the compiler that translates tensor expression from Einstein notation to efficient and portable SQL Einstein summation queries, and the experiments from the paper can be downloaded from https://github.com/ti2-group/sql-einsum. In addition, we provide a website for generating SQL Einstein summation queries at https://sql-einsum.ti2.uni-jena.de.

2 BACKGROUND

In 1916, Einstein introduced a notational convention for tensor expressions that avoids explicit summation signs [18]. By using this notational convention, commonly referred to as Einstein notation, linear algebra or tensor expressions can be represented concisely and clearly. For example, the linear algebra expression $AB^\top v$, where $v \in \mathbb{R}^J$, $A \in \mathbb{R}^{I\times K}$ and $B \in \mathbb{R}^{J\times K}$, reads in Einstein notation as follows: $A_{ik}B_{jk}v_j$. In the original notation, pairs of repeated indices are implicitly used for summation. The indices j and k appear twice in the expression $A_{ik}B_{jk}v_j$, and therefore are used as summation indices. The example expression in Einstein notation is thus a short version of $\sum_j \sum_k A_{ik}B_{jk}v_j$, without the explicit summation signs.

Note that for each tensor within an expression in Einstein notation, the number of indices determines the order of the tensor. In the example expression A_{ik} is a second order tensor, that is, a matrix. Within a tensor, the ℓ-th index refers to its ℓ-th axis. When two tensors share an index, then the two axis must have the same size, that is the same number of elements. In our example expression, the second axis of A and the second axis of B share the same index, namely k, so these two axes must have the same size. Also the first axis of B and the first axis of v share an index, namely j. Therefore, these two axes must also have the same size.

Each tensor expression in Einstein notation can be evaluated as a nested set of for-loops. Suppose we evaluate the example expression and store the result in a vector r, where $r \in \mathbb{R}^I$. Listing 1 shows how to perform this computation in Python using nested for-loops. Note that i is the only index that remains after the computation and is therefore used to index the output vector r.

Listing 1: Computing $\sum_j \sum_k A_{ik}B_{jk}v_j$ using nested for-loops.

```
r = np.zeros(A.shape[0]) # output is a vector
for i in range(A.shape[0]):
  for j in range(B.shape[0]):   # summation over j
    for k in range(A.shape[1]): # summation over k
      r[i] += A[i, k] * B[j, k] * v[j] # i remains
```

In contrast to Einstein's original notation, modern Einstein notation explicitly states the indices that remain after the expression is evaluated. Put differently, this means that modern Einstein summation explicitly specifies the indices for the output tensor. Instead of writing $A_{ik}B_{jk}v_j$ and assuming that j and k are used for summation, because they appear twice in the expression, the example expression in modern Einstein notation reads as follows: $A_{ik}B_{jk}v_j \rightarrow r_i$, where i is the output index. In modern Einstein notation, expressions like $A_{ik}B_{jk}v_j \rightarrow s$ or $A_{ik}B_{jk}v_j \rightarrow R_{ijk}$ are also possible. The first expression evaluates to a scalar, the second to a third order tensor (see Listing 2). In modern Einstein notation the following rule applies: indices that are not part of the output tensor are used for summation. Note that the results of the last two expressions are not directly computable using only the linear algebra notation. Thus, modern Einstein notation significantly increases the expressive power over Einstein's original notation and also over linear algebra notation.

Listing 2: Computing $A_{ik}B_{jk}v_j \rightarrow s$ and $A_{ik}B_{jk}v_j \rightarrow R_{ijk}$ simultaneously using the same set of nested for-loops.

```
s = 0 # output is a scalar
R = np.zeros((A.shape[0], B.shape[0],
   A.shape[1])) # output is a third order tensor
for i in range(A.shape[0]):
  for j in range(B.shape[0]):
    for k in range(A.shape[1]):
      s += A[i, k] * B[j, k] * v[j]
      R[i, j, k] += A[i, k] * B[j, k] * v[j]
```

Common Einstein summation APIs allow both the modern way of expressing Einstein notation with the arrow, and the classical one with the implicit convention of summation over repeated indices. Here, to represent a tensor expression, we stick to the modern way with the arrow to avoid ambiguity. If we do not care about the names of the tensors, then the essential information of a tensor expression can be encoded using just the indices of the tensors in a simple format string. We use a format string similar to that of the `einsum` function in the NumPy library [19]. The format string representing the example expression $A_{ik}B_{jk}v_j \rightarrow r_i$ is `"ik,jk,j->i"`. Table 1

Table 1: Examples of format strings for tensor expressions.

Operation	Format string
Matrix diagonal	`ii->i`
Vector outer product	`i,j->ij`
Mahalanobis distance	`i,ij,j->`
Marginalization (sum over multiple axes)	`ijklmno->m`
Batch matrix multiplication	`bik,bkj->bij`
Bilinear transformation [22]	`ik,klj,il->ij`
Element-wise product of two 4D tensors	`ijkl,ijkl->ijkl`
Matrix chain multiplication	`ik,kl,lm,mn,nj->ij`
2×3-tensor network [23]	`ij,iml,lo,jk,kmn,no->`
Tucker decomposition [24]	`ijkl,ai,bj,ck,dl->abcd`

contains example expressions to illustrate the expressive power of this Einstein-like notation.

Although it is trivial to evaluate expressions in Einstein notation in a brute forced manner, that is, with a set of nested for-loops, it is NP-hard to do so with optimal computational cost [20]. Summing over pairs of repeated indices is called tensor contraction. The computational cost of contracting a tensor expression depends heavily on the contraction order, whereas the result does not [21]. In our example expression $A_{ik}B_{jk}v_j \rightarrow r_i$, there are two possible sequences for pairwise contracting the common indices. Contracting k first and then j results in a matrix-matrix multiplication of A and B, and then the intermediate matrix is multiplied by the vector v. On the other hand, contracting j first and then k leads to a matrix-vector multiplication between B and v, and then matrix A is multiplied with the intermediate vector. The contraction order j, k avoids the expensive matrix-matrix multiplication and is therefore preferable here. Listing 3 shows Python code illustrating the efficient contraction order j, k of the example expression.

Listing 3: Computing $A_{ik}B_{jk}v_j \rightarrow r_i$ efficiently.

```python
r = np.zeros(A.shape[0]) # output is a vector
tmp = np.zeros(A.shape[1]) # intermediate vector
for j in range(B.shape[0]):   # summation over j
  for k in range(A.shape[1]):
    tmp[k] += B[j, k] * v[j]  # matrix * vector
for k in range(A.shape[1]):   # summation over k
  for i in range(A.shape[0]):
    r[i] += A[i, k] * tmp[k]  # matrix * vector
```

The terms Einstein notation and Einstein summation are often used interchangeably. To make the naming in this paper unambiguous, we use the term Einstein notation for the format string representing the tensor expression, and Einstein summation for the actual evaluation of a tensor expression.

3 EINSTEIN SUMMATION IN SQL

In this section, we map the Einstein-like notation presented in the Background section to SQL for enabling Einstein summation in databases. First, we choose a portable schema for representing tensors in SQL, that is, we choose a design for relations and their data types so that tensors can be encoded across various DBMSes. Second, we present mapping rules to generate non-nested Einstein summation queries from arbitrary format strings. Finally, we show how to decompose Einstein summation queries into smaller parts for exploiting efficient contraction sequences.

3.1 Choosing a portable schema for tensors

In order to make Einstein summation portable across different database vendors and SQL dialects, we choose the coordinate (COO) format to represent tensors. The COO format does not use vendor-specific data types such as vectors, matrices or tensors, but only integers and floating point numbers. The COO format explicitly stores the indices, that is the coordinates, for each value of a tensor [25]. For example, the schema for a 3D tensor looks as follows:

```
A(i INT, j INT, k INT, val DOUBLE).
```

Table `A` stores a 3D tensor. Each value (`val`) in tensor `A` can be addressed by specifying the corresponding indices (`i`, `j`, `k`).

The COO format is a sparse tensor storage format, that is, zero values do not need to be stored explicitly [26]. For tensors consisting mainly of zeros, a sparse storage format saves space and time in computations. For dense tensors, that is tensors with few or no zero values, it is rather inefficient. Here, however, we use the COO format for both dense and sparse tensors because we consider it to be the only suitable format for portable Einstein summation in SQL.

3.2 Mapping Einstein summation to SQL

Given the tensors in COO format and a format string in Einstein notation, any Einstein summation operation can be mapped to a single, non-nested SQL query. Suppose we map the example tensor expression from the Background section to SQL. For clarity, we use index names in the format string that are different from the index names of the input tensors. We also color the indices used for summation with the same color. The format string thus looks as follows: `ac,bc,b->a`. Its corresponding input tensors in COO format are `A(i, j, val)`, `B(i, j, val)`, `v(i, val)`. Note that the coloring of the indices of the input tensors is the same as in the format string, although the index names are different. Listing 4, along with some input data, shows how to compute the example tensor expression in SQL and with NumPy in Python.

Listing 4: Einstein summation in Python and SQL.

```python
import numpy as np
A = np.array([[1.0, 0.0], [0.0, 2.0]])
B = np.array([[3.0, 4.0], [5.0, 6.0], [0.0, 7.0]])
v = np.array([8.0, 0.0, 9.0])
print(np.einsum("ac,bc,b->a", A, B, v))
```

```sql
WITH A(i, j, val) AS ( -- matrix A
  VALUES (0, 0, 1.0), (1, 1, 2.0)
), B(i, j, val) AS (   -- matrix B
  VALUES (0, 0, 3.0), (0, 1, 4.0), (1, 0, 5.0),
         (1, 1, 6.0), (2, 1, 7.0)
), v(i, val) AS (      -- vector v
  VALUES (0, 8.0), (2, 9.0)
) SELECT A.i AS i,                              -- R2
         SUM(A.val * B.val * v.val) AS val -- R3
  FROM   A, B, v                                -- R1
  WHERE  A.j=B.j AND B.i=v.i                    -- R4
  GROUP  BY A.i                                 -- R2
```

For mapping any tensor expression in Einstein notation to SQL, the following four rules (R1 – R4) suffice.

R1 All input tensors are enumerated in the FROM clause.

R2 The indices of the output tensor are enumerated in the SELECT clause and the GROUP BY clause.

R3 The new value is the SUM of all values multiplied together.

R4 Indices that are the same among input tensors are transitively equated in the WHERE clause.

In Listing 4, all four rules are applied to map the example tensor expression to SQL (see comments *R1* – *R4*). For some tensor expressions the conditions to apply the rules R2 and/or R4 are not fulfilled. If the output tensor is a scalar, that is, there are no indices in the format string after the arrow, R2 is skipped. If there is no summation in the tensor expression, that is, there are no common indices in the format string before the arrow, R4 is skipped. Skipping R2 results in a query without a GROUP BY clause, skipping R4 omits the WHERE clause in the query.

The format string of the example tensor expression from Listing 4 contains pairs of repeated indices, namely `b` and `c`, therefore, according to R4 they need to be equated in the WHERE clause. Valid tensor expressions can also contain triples, quadruples, etc. of repeated indices. For example, the element-wise product of three vectors (`d,d,d->d`) contains three repeating indices in the input tensors. These three indices must be transitively equated in the WHERE clause. Given the vectors `u(i, val)`, `v(i, val)` and `w(i, val)`, Listing 5 shows the operation in SQL mapped according to R1 – R4.

Listing 5: Transitively equated indices in the WHERE clause.

```
SELECT u.i AS i,                            -- R2
       SUM(u.val * v.val * w.val) AS val    -- R3
FROM   u, v, w                              -- R1
WHERE  u.i=v.i AND u.i=w.i                  -- R4
GROUP  BY u.i                               -- R2
```

Note that the mapping of the element-wise product of three vectors from Einstein notation to SQL in Listing 5 can be further simplified by removing the SUM operation and the GROUP BY clause. The four rules presented guarantee a correct mapping of Einstein notation to SQL, not a mapping with minimal code size. Further checks may be required to simplify the generated SQL code. However, query optimizers are often smart enough to figure out redundant SQL constructs on their own.

3.3 Optimizing contraction order in SQL

Mapping a tensor expression in Einstein notation to a single, non-nested query can lead to suboptimal running times when executing the query, especially for an expression with many tensors. The suboptimal running times of a large Einstein summation query may be caused by the query optimizer's inability to efficiently decompose the query into multiple smaller parts. Common query optimizers know nothing about the contraction order of repeating indices in a tensor expression and thus are unable to exploit it for computations. However, by using common table expressions or subqueries for intermediate tensors, a single large Einstein summation query can be decomposed into smaller parts, forcing the database engine to adhere to a predefined contraction order. By using GROUP BY and aggregation (SUM) in the intermediate computations, query engines have to adhere to the decomposed evaluation order of an Einstein summation query. Listing 6 shows how to decompose the Einstein summation query from Listing 4 into two matrix-vector computations by using an intermediate vector `k`.

Listing 6: Decomposed Einstein summation query.

```
WITH k(i, val) AS (  -- intermediate vektor k
  SELECT B.j, SUM(v.val * B.val)
    FROM v, B WHERE v.i=B.i GROUP BY B.j -- k = vB
) SELECT A.i AS i, SUM(k.val * A.val) AS val
    FROM k, A WHERE k.i=A.j GROUP BY A.i -- Ak
```

We use opt_einsum [27] for computing an efficient contraction order of a format string and its corresponding tensors. We pass the sizes of the tensors to opt_einsum, or set them to default values if they are missing. opt_einsum contains several path finding algorithms, ranging from an exhaustive search for all possible contraction paths to a greedy heuristic approach. By default, opt_einsum selects the best possible algorithm for a format string while trying to keep path finding times below 1 ms [28].

The result of using opt_einsum is a contraction list that contains enough information for generating the decomposed Einstein summation query. Because a decomposition of a large Einstein summation operation consists of multiple smaller Einstein summation operations, we use the mapping strategy from Section 3.2 multiple times to generate the decomposed query.

4 EXPERIMENTS

We demonstrate the practical value of Einstein summation in SQL by solving problems from four different computational domains. Each of the following subsections is self-contained and includes both a problem description and an experimental evaluation. As DBMSes for executing Einstein summation queries and thus solving the problems, we use HyPer, SQLite, and PostgreSQL. HyPer is a column-oriented in-memory DBMS that achieves high performance for both OLTP and OLAP workloads [29]. We use Tableau's publicly available HyPer API version 0.0.13287, which installs both the HyPer DBMS and its corresponding Python interface locally on the machine. SQLite is a lightweight, widely used, open source database engine. We use SQLite version 3.33.0. PostgreSQL is a popular, open source, row-oriented DBMS. We use PostgreSQL version 12.11. We report the performance for these DBMSes in iterations per second, that is, how many times the problem can be solved in one second, where the computation of the next problem starts as soon as the previous one finishes. In HyPer, we measure two possible query execution modes, compilation and interpretation. We report only the best results of these two modes. For better comparison, the measurements also include the performance of opt_einsum. opt_einsum is a Python package for efficient Einstein summation of large tensor expressions. opt_einsum supports various `einsum` backends. To the best of our knowledge, sparse tensors are not supported in the various `einsum` backends. We use opt_einsum version 3.3.0 with a NumPy backend. At each problem instance in the experiments, the SQL implementations and opt_einsum use identical tensor contraction sequences. We pass a precomputed contraction sequence as an optional argument to opt_einsum, so that opt_einsum can immediately begin to compute the result tensor without having to optimize the contraction path of the tensor expression first.

All following experiments are performed on a machine with an Intel i9-10980XE 18-core processor running Ubuntu 20.04.1 LTS with 128 GB of RAM. Each core has a base frequency of 3.0 GHz and a max turbo frequency of 4.6 GHz and supports the AVX-512 vector instruction set.

4.1 Querying triplestore data

A triplestore is a type of graph database used to store and retrieve triples through semantic queries. A triple is a data entity consisting of three terms (*subject-predicate-object* or *s-p-o* for short), like *Alice-knows-Bob* or *Bob-plays-piano*. A semantic query on these two example triples could be as follows: *Does Alice know anyone who plays the piano?* The answer here would be *yes* (Bob). Semantic queries are usually formulated in SPARQL [30], the standard query language for triplestores. SPARQL queries can be mapped to Einstein summation when the triplestore data is organized as a one-hot encoded third order tensor $T^{n \times n \times n}$, where n is the number of distinct terms in the triples [14, 31].

For our SQL triplestore experiment we use the 120 years of Olympic history dataset [32, 33]. This dataset consists of 1 781 625 triples and contains a total of $n = 544\,171$ distinct terms in the triples. Because each triple is only a data point in the tensor (with a value of one), the tensor is hypersparse (10^{-11} percent of the values are non-zero). The tensor T would need about 500 petabytes in a dense representation. Thus, executing SPARQL queries with Einstein summation requires support for sparse tensors. However, in constrast to our SQL Einstein summation, current Einstein summation implementations do not support sparse tensors. Therefore, we use here the Python package RDFLib (version 6.2.0) instead of opt_einsum for the performance comparison with SQL. RDFLib represents the semantic data as a graph and allows SPARQL queries to be executed on it. We use the SPARQL query in Listing 7 to compare the performance between DBMSes and RDFLib.

Listing 7: SPARQL query for performance comparison. *List all athletes who have won a gold medal and the number of gold medals they have won, in descending order.*

```
PREFIX walls: <http://.../olympics/>
PREFIX rdfs:  <http://.../rdf-schema#>
SELECT ?name (COUNT(?name) AS ?count)
WHERE {
  ?instance walls:athlete ?athlete .          # TP1
  ?instance walls:medal <http://.../Gold> .  # TP2
  ?athlete rdfs:label ?name .                 # TP3
} GROUP BY ?name ORDER BY DESC(?count)
```

To query the Olympic history dataset in SQL with Einstein summation, we convert the dataset to a sparse one-hot encoded third order tensor T. Each of the three triple patterns (`TP1` – `TP3`) in the `WHERE` condition of the SPARQL query of Listing 7 represents a slice of T. To obtain the corresponding slice from a triple pattern, terms are replaced by their IDs and variables are replaced by the slice placeholder ':'. Let a be the ID for `walls:athlete`, m for `walls:medal`, g for `<http://.../Gold>`, and l for `rdfs:label`, the triple patterns yield the following three slices: $T_1 = T[:, a, :]$, $T_2 = T[:, m, g]$ and $T_3 = T[:, l, :]$. The SPARQL query can now be answered with the following tensor expression: `ij,i,jk->k`, where the input tensors correspond to slices T_1, T_2 and T_3 (see Listing 8 for the full SQL query). Figure 2 shows the performance of the full SQL query, that is, computing the slices of T, contracting them with Einstein summation, and sorting the results in descending order.

Listing 8: SQL query for the SPARQL query from Listing 7.

```
WITH T1(i, j, val) AS (     -- T[:,a,:]
  SELECT s, o, val FROM T WHERE p=a
), T2(i, val) AS (          -- T[:,m,g]
  SELECT s, val FROM T WHERE p=m AND o=g
), T3(i, j, val) AS (       -- T[:,l,:]
  SELECT s, o, val FROM T WHERE p=l
), K1 AS (                  -- Einstein summation
  SELECT T1.j AS i, SUM(T2.val * T1.val) AS val
    FROM T2, T1 WHERE T2.i=T1.i GROUP BY T1.j
) SELECT T3.j AS i, SUM(K1.val * T3.val) AS val
    FROM K1, T3 WHERE K1.i=T3.i GROUP BY T3.j
      ORDER BY val DESC
```

Surprisingly, all three DBMSes perform better than RDFLib, with HyPer particularly standing out for its performance. Unlike relational DBMSes, triplestores are optimized for good performance on querying semantic data. However, the results suggest that, at least for medium-sized semantic datasets, relational databases may be a possible alternative to triplestores, as querying the data with Einstein summation shows satisfying performance. Note that we did not index the tensor T in the experiments. Indexing T could further improve the performance of querying semantic data in SQL.

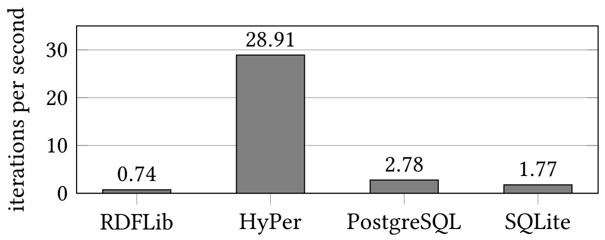
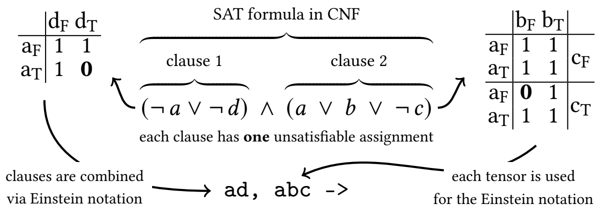


Figure 2: Performance for answering the SPARQL query of Listing 7 on multiple DBMSes and RDFLib (Python).

4.2 Solving SAT problems

SAT, which decides whether a given Boolean formula is satisfiable, is probably the best known NP-complete problem. Satisfiable means that there is an assignment of truth values to the Boolean variables in the formula that evaluates the entire formula as true. Commonly, SAT formulas are represented in conjunctive normal form (CNF). A formula in CNF is a conjunction of clauses, where a clause is a disjunction of literals and a literal is a variable or its negation. Any decision problem in the complexity class NP can be reduced to the SAT problem for CNF formulas [34].

Here, we use Einstein summation in SQL to count the number of solutions to a given SAT formula in CNF. Counting the number of solutions is known as the #SAT problem, which is #P-complete [35]. We adapt the approach of Biamonte et al. [15] to convert a SAT formula in CNF to a tensor network. However, unlike Biamonte et al. we avoid creating a COPY-tensor for each variable of the SAT formula, thus simplifying the computation. The conversion starts by



SAT formula in CNF
clause 1
clause 2
$(\neg a \vee \neg d) \wedge (a \vee b \vee \neg c)$
each clause has **one** unsatisfiable assignment
clauses are combined via Einstein notation
ad, abc ->
each tensor is used for the Einstein notation

Figure 3: Conversion of a SAT formula into an Einstein summation problem to compute the number of its solutions.

creating a tensor for each clause of the SAT formula. The order of a tensor corresponds to the number of variables in the clause. The size of each dimension of the tensor is two. For a clause with two variables, this would result in a $\{0,1\}^{2\times2}$ tensor; with three variables, it would result in a $\{0,1\}^{2\times2\times2}$ tensor. Each dimension of a tensor corresponds to a variable of the clause. Thus, the coordinates of a single point in the tensor describe a possible assignment of truth values to the Boolean variables in the clause. If an assignment satisfies the clause, then the value of the point is one, otherwise it is zero. Note that in each tensor only one value is zero and all others are one, because in a clause the variables are disjointly connected and thus only one variable assignment does not satisfy the clause. Finding the assignment that does not satisfy the clause is computationally cheap, because it only requires to ensure that each variable and its possible negation evaluates to false. Finally, the tensor network combines the tensors in such a way that the indices of the tensors correspond to the variables in the clauses. Figure 3 shows the conversion of an example SAT formula in CNF into an Einstein summation problem. Listing 9 computes in SQL the number of solutions for the example SAT formula.

Listing 9: SQL query for computing the number of solutions for the SAT formula $(\neg a \vee \neg d) \wedge (a \vee b \vee \neg c)$.

```
WITH T1(i, j, val) AS (  -- tensor for clause 1
  VALUES (0, 0, 1), (0, 1, 1), (1, 0, 1)
), T2(i, j, k, val) AS ( -- tensor for clause 2
  VALUES (0, 0, 0, 1), (0, 1, 0, 1),
         (0, 1, 1, 1), (1, 0, 0, 1), (1, 0, 1, 1),
         (1, 1, 0, 1), (1, 1, 1, 1)
) SELECT SUM(T1.val * T2.val) AS val FROM T1, T2
    WHERE T1.i=T2.i       -- Einstein summation
```

To explore the practicality of solving SAT problems in SQL, we studied the package management tool Anaconda [36]. With Anaconda, packages can be installed and managed for the Python and R programming languages. When a new package is installed, Anaconda verifies dependencies between already existing packages and the package to be installed. For this purpose, Anaconda internally creates a CNF formula, which it checks for satisfiability. This formula encodes in its clauses the packages and the mutual dependencies. For our use case, we run the Anaconda command `conda install sqlite` on a newly created conda environment. In this scenario, conda has to solve a CNF formula with 718 clauses and 378 variables. This formula is given as a 3-SAT problem and can therefore be converted to a tensor network using the approach described above. In 3-SAT, each clause uses at most three literals. Note that there can only be a maximum of 14 unique tensors in a 3-SAT problem: two for clauses with one variable, four for clauses with two variables, and eight for clauses with three variables. In the SQL queries and the opt_einsum computations in Python, we use only the required subset of the 14 tensors to compute the number of solutions for a 3-SAT formula. Reusing tensors avoids the need to create a separate tensor for each clause. For our measurements, we vary the number of clauses to get a sense of the scalability of solving SAT problems in SQL. The performance of evaluating the tensor networks for different numbers of clauses is shown in Figure 4.

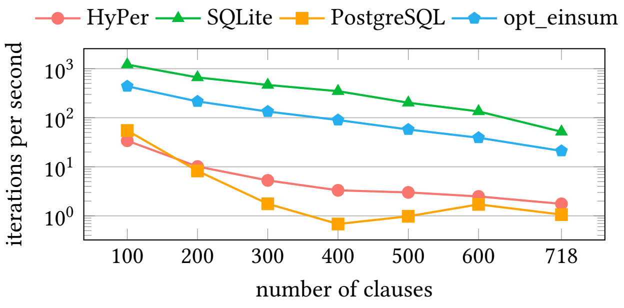


Figure 4: Performance based on the number of clauses in a CNF formula of a 3-SAT problem.

The performance measurements are shown on a logarithmic scale. As expected, opt_einsum outperforms HyPer and PostgreSQL because it is a dense tensor algebra computation problem with relatively small tensors, where a SQL query can easily exceed 100 KB in size. However, for this kind of problem SQLite shows even better performance than opt_einsum. With SQLite the Einstein summation for the complete SAT problem (718 clauses) can be evaluated 52 times per second, but with opt_einsum only 21 times per second. While experimenting with different SAT formulas, we also reached the limits of the maximum allowed dimensions in NumPy (32 dimensions) when running some SAT problems and therefore could not perform the computations with opt_einsum. With SQLite, the limit on the number of dimensions depends on the compile-time parameter `SQLITE_MAX_COLUMN`, which can be set to values up to 32767. Thus, we were able to solve more problems with SQLite than with opt_einsum.

4.3 Inference in graphical models

Graphical models are sparse representations of multivariate probability distributions, where nodes represent random variables and edges encode conditional dependencies (interactions) among the variables. Inference in graphical models means computing probabilities of the form P(X=x|E=e), where X is a set of query variables and E is a set of evidence variables. The choice of X and E is arbitrary, which makes graphical models very flexible machine learning models.

Here, we use as an example the breast cancer dataset from the UCI machine learning repository [37], in which ten variables such as age, tumor size, the degree of malignancy, and whether a tumor has recurred were collected from 286 individuals. After learning a graphical model for this dataset, we can ask inference queries like *What is the probability of breast cancer recurrence given a certain age and the fact that the person was irradiated?* or *Given the number* *of axillary lymph nodes containing metastatic breast cancer and the grade of malignancy, what tumor size is most likely?*. For answering questions like these, the graphical model must be turned into a data structure that supports inference queries. It has been shown recently [16] that tensor networks can serve as such a data structure. Inference queries on graphical models represented by tensor networks are performed via Einstein summation.

Here, we use the Python package `cgmodsel` [38] to learn a graphical model for the breast cancer dataset. The interactions of the learned model are given by a symmetric block-sparse matrix Q. The non-zero blocks in Q correspond to edges in the graphical model and are used, as shown in Figure 5, as tensors of the tensor network representation.

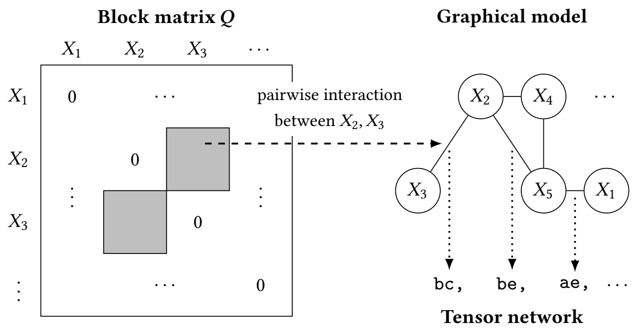


Figure 5: Translation of the pairwise interaction matrix Q to a tensor network. The blocks of Q are non-zero exactly when there is an edge between the two underlying variables in the graphical model (dashed arrow). Each edge of the graphical model becomes a tensor in the tensor network (dotted arrow).

The tensor network for the breast cancer dataset has 21 matrices with shapes ranging from $\mathbb{R}^{2\times3}$ to $\mathbb{R}^{11\times7}$. For the SQL experiment, we store the matrices in the DBMS before running the queries. Using these matrices, we compute the probability of breast cancer recurrence considering all the patient's data as evidence, that is, we compute P(recurrence|evidence). The patient's data can be embedded in the query as one-hot encoded vectors. However, instead of computing the probability for a single patient, we compute the probabilities for multiple patients simultaneously, that is, we embed one-hot encoded matrices as evidence in the query instead of vectors. Figure 6 shows the performance of computing probabilities with different numbers of patients (batch size) in the query.

The graphical model query executes significantly faster in opt_einsum than in the three DBMSes, for all batch sizes. However, especially on HyPer the performance for the graphical model query is satisfying, also in terms of scalability for the batch size. In SQLite and PostgreSQL, the performance degrades faster with increasing batch size than in HyPer and NumPy. But for graphical model inference with small batch sizes, DBMSes are fast enough in practice.

4.4 Simulating quantum circuits

Quantum circuits are commonly used to describe quantum computations on qubits. A qubit is the smallest possible unit of quantum information. A single qubit is a quantum system with two states,

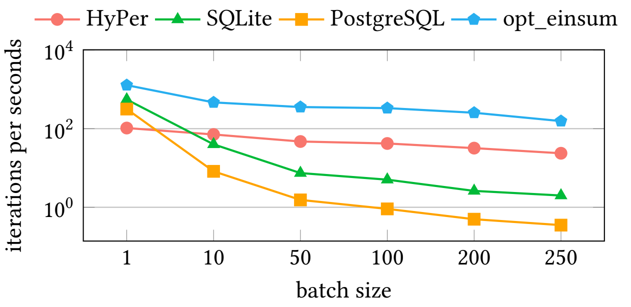


Figure 6: Performance based on the number of patients (batch size) in the graphical model query.

but unlike its classical counterpart, the bit, its value is not discrete but continuous due to the principle of quantum superposition. Quantum circuits basically consist of quantum gates connected by quantum wires. Quantum gates are single or double qubit operators representing unitary transformations on qubits. Figure 7 shows an example quantum circuit for a two-qubit system consisting of two Hadamard gates (single qubit operators) and one CX gate (double qubit operator). The output of a quantum circuit is not a bit string, but a complex probability distribution. Only after the output is measured, the value for a given qubit becomes either 0 or 1.

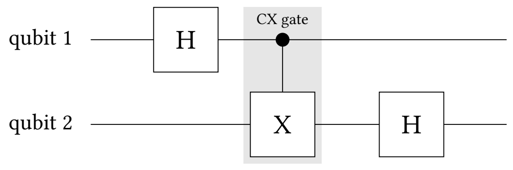


Figure 7: Simple two-qubit quantum circuit.

Because reliable quantum computers are still in their infancy, simulating quantum circuits on classical computers is the usual way to perform quantum computations. On a laptop it is feasible to simulate quantum circuits with about 30 qubits. However, it takes a supercomputer to simulate 56 qubits [39]. Efficient simulation is key to solving larger quantum problems on classical machines. Tensor networks have proven to be an efficient simulation backend for quantum computations [40–42]. It is straightforward to map quantum circuits onto tensor networks [17]. For our example circuit from Figure 7, the format string `a,b,ca,dbc,ed->ce` represents the Einstein summation operation to compute the corresponding probability distribution. In the format string, `"a,b"` represents the two input qubits. The input qubits are vectors of size two that encode the initial state of the circuit. The symbol $|1\rangle$ stands for the vector $(1,0)$ and the symbol $|0\rangle$ for the vector $(0,1)$, so that for the example circuit four possible initial states, namely $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, can be encoded using two qubits. The two Hadamard gates, which are essentially 2×2-matrices, are represented by `"ca,ed"`. The only peculiarity that must be taken into account to execute the example circuit with Einstein summation is that commonly the CX gate is represented by a 4×4-matrix. In Einstein summation, however, the CX gate is instead a 2×2×2-tensor. The CX gate is represented by `"dbc"` in the format string. Finally, the 2×2-matrix `"ce"` describes the probabilities of the four possible output states that the example circuit computes.

Quantum physics problems require the use of complex numbers [43, 44]. Therefore, to enable quantum computations in SQL, we extend the decomposed Einstein summation queries from Subsection 3.3 to support complex numbers. Unfortunately, complex numbers are not part of the SQL standard [45] and are therefore not natively supported by most DBMSes. However, complex numbers can be realized with SQL in a portable way [1]. But, when multiplying complex numbers, the boundaries between the real and imaginary parts must be crossed. Multiplying two complex numbers $a+bi$ and $c+di$, results in the complex number $(ac-bd)+(ad+bc)i$. Fortunately, the decomposed Einstein summation queries perform only multiplication between two numbers, because each common table expression uses at most two tensors. Therefore, to enable Einstein summation with complex numbers in SQL, it is sufficient to hardcode the formula for multiplying two complex numbers.

For our simulation experiments of quantum circuits, we use parts of Google's Sycamore quantum supremacy circuit [46]. The full Sycamore quantum supremacy circuit contains 53 qubits and has a depth of 20, meaning it contains 20 cycles of unitary operations. Using the Julia package Yao [47], we can generate different quantum circuits instances with different numbers of qubits and different depths of the Sycamore circuit. The Yao package also allows us to convert these circuits to Einstein notation.

For the first experiment we set the number of qubits to ten and vary the depth of the circuits (see Figure 8). With increasing circuit depth and a constant number of qubits, all implementations except PostgreSQL show satisfying performance scaling. Considering the fact that NumPy natively supports complex numbers and we have to introduce significant overhead in SQL to enable them, it is surprising that SQLite's performance is only slightly worse compared to opt_einsum with the NumPy backend.

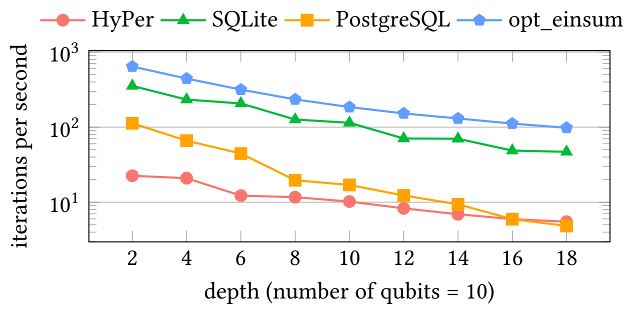


Figure 8: Performance for computing different quantum circuits with changing depth and fixed number of qubits.

However, with varying number of qubits and a constant depth, the measured performance diverges more for the databases (see Figure 9). For a small number of qubits, SQLite outperforms even opt_einsum, but for many qubits, SQLite's performance drops under HyPer. A large number of qubits means that the dense output tensor describing the complex probability distribution is also high-dimensional. It is rather inefficient to represent dense high-dimensional tensors in SQL by using a sparse tensor format, therefore opt_einsum clearly outperforms here the database implementations of quantum circuits with many qubits.

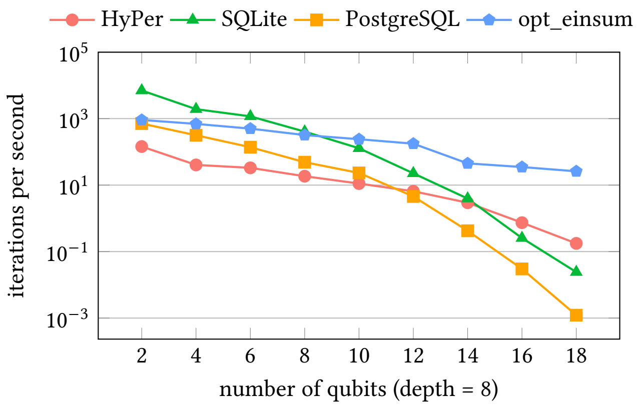


Figure 9: Performance for computing different quantum circuits with changing number of qubits and fixed depth.

5 DISCUSSION: THE ROLE OF QUERY ENGINES IN EINSTEIN SUMMATION

Performing Einstein summation with hundreds or even thousands of tensors presents new challenges to query engines and their query optimizers. Large Einstein summation queries have some interesting properties. They often do not require much data, but perform extensive computations. This is not the standard DBMS use case, where computations are performed on large datasets. Nevertheless, computations such as large Einstein summations are used in practice, and surprisingly, some DBMSes, even if not tuned for these types of computations, perform quite well. Why is it that one DBMS finishes the query in a fraction of a second, whereas another DBMS does not finish the query at all?

For large Einstein summation queries, the bottleneck is often not executing the query, but optimizing it. However, because Einstein summation queries are already decomposed into sequences of pairwise contractions, the additional query optimizations are mostly superfluous and only cost computational time with no additional benefit. For this kind of queries it would be sufficient to analyze only which contractions, that is, which common table expressions, can be executed at the same time. Finding independent computations (common table expressions) that can be executed concurrently is a rather lightweight optimization and would require only a small amount of computational time. However, superfluous optimizations for computationally intensive queries are common among DBMSes.

To demonstrate that query optimization can be the bottleneck for such queries, we use as an example a SAT problem with 952 clauses. We also include DuckDB [48] (version 0.5.0) in our measurements, as it demonstrates particularly well that excessive query optimization can be the bottleneck for queries with little data but large computations. To measure the time required for query optimization, we measure the time to determine a query plan. We then subtract the time needed to compute the query plan from the total runtime of the query to obtain only the execution time.

Table 2 shows the planning and execution times for the example SAT problem. It can be seen that the planning time in SQLite is marginal, whereas the planning time in HyPer takes almost the entire time of the query. Note that here we measure the time for

DBMS	Planning time	Execution time
opt_einsum (NumPy backend)	0.00 s	3.69 s
SQLite	0.03 s	0.37 s
HyPer (interpreted)	0.87 s	0.08 s
PostgreSQL	1.51 s	20.19 s
DuckDB	*N/A*	*N/A*
DuckDB (no optimizations)	0.20 s	0.97 s

Table 2: Planning and execution times for the SAT problem.

HyPer in interpretation mode, in compilation mode the bottleneck would be the compilation of the large SAT query (169 KB). However, considering only the query execution (without planning), HyPer has the best performance for the example SAT query. Interestingly, for this problem, opt_einsum is much slower than the DBMSes, except PostgreSQL. In DuckDB, the query plan for the example SAT problem was still not computed after five hours, therefore we terminated the computation. However, DuckDB supports disabling query optimizations by setting a pragma before executing the query. With query optimizations disabled, DuckDB showed also satisfying performance. Most DBMSes do not allow disabling query optimizations in a simple way. The ability to disable query optimizations before executing a query is a highly practical feature and should therefore be firmly established in modern DBMSes.

Is disabling query optimizations the best strategy for efficient in-DBMS Einstein summation? Would it not be better to create efficient query plans by the query optimizer itself instead of hard-coding them externally in the query? Einstein summation problems are often repetitive, meaning that only one or a few parameter tensors and/or data tensors change so that the contraction path of an entire expression remains the same. Forcing a DBMS to optimize the contraction path on each execution of a query would be redundant. However, caching the query plans could avoid redundant computations of contraction paths. Still, finding efficient contraction paths is a hard problem. We used opt_einsum's built-in contraction path optimizer, but for example cotengra [49] in combination with KaHyPar [50] can find more efficient contraction paths, but often at the price of longer computation time. Such design decisions about which optimizer to use for contraction path computation are easily replaceable if contraction paths are computed outside the DBMS. Moreover, such optimizers can be more efficient than traditional query optimizers because the problem they optimize, namely the contraction path, is known in advance. However, the DBMS knows better the underlying dimensions and sparsity levels of the data. Also, traditional contraction path optimizers do not consider sparsity, and the specialized implementations that do adapt approaches from the DBMS community [14], such as worst-case optimal join algorithms [51]. Therefore, optimizing contraction paths for computations with sparse tensors is better done by the query optimizer than externally. For dense tensors, the decision between external path optimization versus in-DBMS path optimization remains unclear. Note that, in the presented approach, we decompose a single, non-nested Einstein summation query into a large number of nested pairwise Einstein summation operations, that is, we replace a single GROUP BY clause in the execution tree with multiple GROUP BY clauses, thereby splitting and deferring joins and summation. Such query optimizations are well known [52–54] – but not widely supported by current DBMSes. Therefore, forcing an efficient evaluation order of the Einstein summation externally, makes the query more performance portable compared to relying on the supported features of query optimizers. Nevertheless, naive Einstein summation in SQL, that is, without forcing an efficient evaluation order externally, shows that blindly executing joins before executing GROUP BY is an inefficient query optimization strategy for performing such aggregational computations. Modern query optimizers should be able to defer joins in such scenarios, but surprisingly, most do not.

6 RELATED WORK

Intensive work has been done to incorporate machine learning, data mining, artificial intelligence, and statistical routines into relational databases [55]. The overall idea of the different approaches is to leverage database queries to trigger in-DBMS analytics. Approaches to support analytical queries range from pure SQL solutions [3, 7, 56, 57], to modifying the language and thus the internals of a database system [5, 58, 59], or even, to building new database systems with better support for analytical tasks [60–62]. In between are approaches that exploit the standard database extension mechanisms such as user-defined types, functions and aggregates [63–66].

The mapping of Einstein summation to standard SQL presented in this paper is a contribution to pure SQL analytics. Note, however, that the above approaches mostly focus on model training, that is, implementing iterative algorithms for learning parameters of various statistical models such as regression, classification, or clustering. While in-DBMS model training is proving very useful in avoiding copying huge amounts of data from one system to another, there is also, as recently noted in the Seattle report on database research [67], an immediate need for efficient in-DBMS inference. Einstein summation naturally allows serving inference queries on various AI models (Boolean, semantic, probabilistic). These inference queries are manifestations of tensor networks, that is, multiplicative tensor expressions with many high-dimensional tensors. Using tensor networks for analytics is an emerging field [68–72], with many applications yet to be found.

7 CONCLUSIONS

In this paper we presented mapping rules to translate tensor expressions given in an Einstein-like notation to SQL. The generated Einstein summation queries are highly portable between different DBMSes and exploit efficient contraction sequences in their execution. Our experiments demonstrate that Einstein summation queries offer new opportunities to process data in SQL. The performance of Einstein summation queries is satisfying across different DBMSes. On a modern in-memory DBMS like HyPer, Einstein summation queries for large sparse problems can even outperform specialized libraries. For dense problems with hundreds of tensor contractions over small tensors, SQLite surprisingly shows superior performance over other DBMSes and occasionally even over opt_einsum with a NumPy backend. However, besides of broadening the range of practical problems that can be solved with database queries, Einstein summation in SQL also offers the possibility of improving query engines by providing contraction algorithms, practical query decomposition strategies, and challenging SQL queries.

REFERENCES

[1] D. Marten, H. Meyer, and A. Heuer, “Calculating fourier transforms in SQL,” in *ADBIS*, 2019.

[2] M. E. Schüle, A. Kemper, and T. Neumann, “Recursive sql for data mining,” in *SSDBM*, 2022.

[3] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis, “Machine learning, linear algebra, and more: Is SQL all you need?,” in *CIDR*, 2022.

[4] T. Fischer, D. Hirn, and T. Grust, “Snakes on a plan: Compiling python functions into plain SQL queries,” in *SIGMOD*, 2022.

[5] M. E. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günnemann, and T. Neumann, “In-database machine learning: Gradient descent and tensor algebra for main memory database systems,” in *BTW*, 2019.

[6] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine, “Scalable linear algebra on a relational database system,” *IEEE Trans. Knowl. Data Eng.*, 2019.

[7] D. Hirn and T. Grust, “One WITH RECURSIVE is worth many GOTOs,” in *SIGMOD*, 2021.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, *et al.*, “Array programming with NumPy,” *Nature*, 2020.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” 2019.

[11] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” 2018.

[12] Matthew Rocklin, “Dask: Parallel Computation with Blocked algorithms and Task Scheduling,” in *Proceedings of the 14th Python in Science Conference*, 2015.

[13] R. Nishino and S. H. C. Loomis, “Cupy: A numpy-compatible library for nvidia gpu calculations,” *Workshop on machine learning systems (LearningSys) in Neural Information Processing Systems (NIPS)*, 2017.

[14] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, and A. N. Ngomo, “Tentris - A tensor-based triple store,” in *ISWC*, 2020.

[15] J. D. Biamonte, J. Morton, and J. W. Turner, “Tensor network contractions for #sat,” *Journal of Statistical Physics*, 2015.

[16] E. Robeva and A. Seigal, “Duality of graphical models and tensor networks,” *CoRR*, vol. abs/1710.01437, 2017.

[17] I. L. Markov and Y. Shi, “Simulating quantum computation by contracting tensor networks,” *SIAM J. Comput.*, 2008.

[18] A. Einstein, “The foundation of the general theory of relativity,” *Annalen der Physik*, 1916.

[19] O. Bilaniuk, “Einstein summation in numpy.” https://obilaniu6266h16.wordpress.com/2016/02/04/einstein-summation-in-numpy/, 2016.

[20] C. Lam, P. Sadayappan, and R. Wenger, “On optimizing a class of multi-dimensional loops with reductions for parallel execution,” *Parallel Process. Lett.*, 1997.

[21] F. Schindler and A. S. Jermyn, “Algorithms for tensor network contraction ordering,” *Machine Learning: Science and Technology*, 2020.

[22] Torch Contributors, “Bilinear.” https://pytorch.org/docs/stable/generated/torch.nn.Bilinear.html, 2019.

[23] J. Jakes-Schauer, D. Anekstein, and P. Wocjan, “Carving-width and contraction trees for tensor networks,” *arXiv*, 2019.

[24] E. Robeva and A. Seigal, “Duality of graphical models and tensor networks,” *Information and Inference: A Journal of the IMA*, 2019.

[25] D. Marten, H. Meyer, D. Dietrich, and A. Heuer, “Sparse and dense linear algebra for machine learning on parallel-rdbms using SQL,” *Open J. Big Data*, 2019.

[26] S. Chou, F. Kjolstad, and S. P. Amarasinghe, “Format abstraction for sparse tensor algebra compilers,” *Proc. ACM Program. Lang.*, 2018.

[27] D. G. A. Smith and J. Gray, “opt_einsum - A python package for optimizing contraction order for einsum-like expressions,” *J. Open Source Softw.*, 2018.

[28] D. G. A. Smith, “opt_einsum docs.” https://optimized-einsum.readthedocs.io, 2018.

[29] A. Kemper and T. Neumann, “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots,” in *ICDE*, 2011.

[30] S. Harris and A. Seaborne, “Sparql 1.1 query language,” *W3C*, 2013.

[31] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, and A. N. Ngomo, “Extended example on Tentris.” https://tentris.dice-research.org/iswc2020/, 2020.

[32] A. Addlesee, “Creating linked data.” https://medium.com/wallscope/creating-linked-data-31c7dd479a9e. Accessed: 2022-08-04.

[33] R. Griffin, “120 years of olympic history: athletes and results.” https://www.kaggle.com/datasets/heesoo37/120-years-of-olympic-history-athletes-and-results. Accessed: 2022-08-04.

[34] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[35] L. G. Valiant, “The complexity of computing the permanent,” *Theor. Comput. Sci.*, 1979.

[36] “Anaconda software distribution,” 2020.

[37] D. Dua and C. Graff, “UCI machine learning repository,” 2017.

[38] F. Nussbaum and J. Giesen, “Pairwise sparse + low-rank models for variables of mixed type,” *J. Multivar. Anal.*, 2020.

[39] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, E. W. Draeger, E. T. Holland, and R. Wisnieff, “Pareto-efficient quantum circuit simulation using tensor contraction deferral,” *arXiv*, 2017.

[40] D. Liakh and USDOE, “Exatensor. computer software,” 2019.

[41] B. Villalonga, S. Boixo, B. Nelson, C. Henze, E. Rieffel, R. Biswas, and S. Mandrà, “A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware,” *npj Quantum Information*, 2019.

[42] F. Pan, K. Chen, and P. Zhang, “Solving the sampling problem of the sycamore quantum circuits,” *Phys. Rev. Lett.*, 2022.

[43] M.-O. Renou, D. Trillo, M. Weilenmann, T. P. Le, A. Tavakoli, N. Gisin, A. Acín, and M. Navascués, “Quantum theory based on real numbers can be experimentally falsified,” *Nature*, 2021.

[44] M.-C. Chen, C. Wang, F.-M. Liu, J.-W. Wang, C. Ying, Z.-X. Shang, Y. Wu, M. Gong, H. Deng, F.-T. Liang, *et al.*, “Ruling out real-valued standard formalism of quantum theory,” *Physical Review Letters*, 2022.

[45] ISO/IEC 9075-2:2016, *Database languages – SQL – Part 2: Foundation.* 2016.

[46] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, *et al.*, “Quantum supremacy using a programmable superconducting processor,” *Nature*, 2019.

[47] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, “Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design,” *Quantum*, 2020.

[48] M. Raasveldt and H. Mühleisen, “Duckdb: an embeddable analytical database,” in *SIGMOD*, 2019.

[49] J. Gray and S. Kourtis, “Hyper-optimized tensor network contraction,” *Quantum*, 2021.

[50] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, “k-way hypergraph partitioning via n-level recursive bisection,” in *ALENEX*, 2016.

[51] H. Q. Ngo, C. Ré, and A. Rudra, “Skew strikes back: new developments in the theory of join algorithms,” *SIGMOD Rec.*, 2013.

[52] S. Chaudhuri and K. Shim, “Including group-by in query optimization,” in *VLDB*, 1994.

[53] W. P. Yan and P. Larson, “Performing group-by before join,” in *ICDE*, 1994.

[54] M. Eich, P. Fender, and G. Moerkotte, “Efficient generation of query plans containing group-by, join, and groupjoin,” *VLDB J.*, 2018.

[55] M. Boehm, A. Kumar, and J. Yang, *Data Management in Machine Learning Systems.* 2019.

[56] D. Marten and A. Heuer, “Machine learning on large databases: Transforming hidden markov models to SQL statements,” *Open J. Databases*, 2017.

[57] L. Du, “In-machine-learning database: Reimagining deep learning with old-school SQL,” *arXiv*, 2020.

[58] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao, “Declarative recursive computation on an RDBMS,” *Proc. VLDB Endow.*, 2019.

[59] M. E. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Günnemann, “In-database machine learning with SQL on gpus,” in *SSDBM*, 2021.

[60] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas, “The monte carlo database system: Stochastic analysis close to the data,” *ACM Trans. Database Syst.*, 2011.

[61] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine, “Simulation of database-valued markov chains using simsql,” in *SIGMOD*, 2013.

[62] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine, “Scalable linear algebra on a relational database system,” in *ICDE*, 2017.

[63] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, “MAD skills: New analysis practices for big data,” *Proc. VLDB Endow.*, 2009.

[64] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, “The madlib analytics library or MAD skills, the SQL,” *Proc. VLDB Endow.*, 2012.

[65] X. Feng, A. Kumar, B. Recht, and C. Ré, “Towards a unified architecture for in-rdbms analytics,” in *SIGMOD*, 2012.

[66] Y. Cheng, C. Qin, and F. Rusu, “GLADE: big data analytics made easy,” in *SIGMOD* (K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, eds.), 2012.

[67] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. A. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, *et al.*, “The seattle report on database research,” *Commun. ACM*, August 2022.

[68] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, “Tensorizing neural networks,” in *Neural Information Processing Systems (NIPS)*, 2015.

[69] E. M. Stoudenmire and D. J. Schwab, “Supervised learning with tensor networks,” in *Neural Information Processing Systems (NIPS)*, 2016.

[70] S. Cheng, L. Wang, T. Xiang, and P. Zhang, “Tree tensor networks for generative modeling,” *Phys. Rev. B*, 2019.

[71] W. Huggins, P. Patil, B. Mitchell, K. B. Whaley, and E. M. Stoudenmire, “Towards quantum machine learning with tensor networks,” *Quantum Science and Technology*, 2019.

[72] I. Glasser, N. Pancotti, and J. I. Cirac, “From probabilistic graphical models to generalized tensor networks for supervised learning,” *IEEE Access*, 2020.