In-Database TensorDB Clients and Operators References

Mijung Kim

June 17, 2014

TensorDB is a data management system based on tensor-relational model that supports tensor-relational operations [4]. In-database TensorDB focuses on building in-database implementation of the tensor-relational operations in an array database, SciDB [1]. As TensorDB clients, we provide static and dynamic tensor decompositions.

For a static tensor decomposition, we consider an alternating least squares (ALS) based implementation of CP decomposition [2, 3]. While we leverage some of the operations in SciDB, most of the operations involved in implementing the CP decomposition in an array database are not available in common array databases. We provide chunk-based implementations of the various operations involved in the CP decomposition.

For a dynamic tensor decomposition, we adapt the Dynamic Tensor Analysis (DTA) algorithm [6] for in-database operation. Note that, unlike CP, DTA assumes a dense core matrix as in Tucker decomposition [7]. DTA incrementally maintains covariance matrices for each mode and computes factor matrices by taking the leading eigen-vectors of the covariance matrices.

The in-database CP and DTA algorithm are implemented using the TensorDB operators along with SciDB operators. The TensorDB operators are chunk-based tensor operators (*matricization, Khatri-Rao product, Hadamard product, normalization*, and *copyArray* operators, etc.) needed for implementing in-database tensor decompositions. Each of these leverages the chunk ordering and access mechanism.

In addition to the above chunked operators, we also implement two non-chunked operators, *pseudoinverse* and *eigen-decomposition*. While these require their inputs to fit into the memory, since (during tensor decomposition) inputs are often relatively small matrices, this rarely constitutes a problem.

The in-database TensorDB clients for tensor decomposition operations are made of a set of Python scripts. The TensorDB clients run a series of in-database TensorDB operations and SciDB operations for tensor decomposition operations. The guide for the package installation of SciDB and running SciDB are referred to in [1]. The TensorDB package is built on the SciDB 12.12 package [1]. We present the references of the TensorDB clients and operators next.

1 In-Database TensorDB Clients

1.1 `cp_als.py`

Description Given the input tensor, run the CP ALS (Alternating Least Squares) algorithm

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target rank, e.g.,: 10
- matricized_modes (optional): modes which use materialized matricization, e.g., 1,2 - mode-1 and 2 use the materialized matricization
- max_iter (optional): the maximum iteration count, default value: 50
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The output includes factor matrices and a core. The factor matrices are *tensor_name*_fac_0, *tensor_name*_fac_1, ..., and the core is *tensor_name*_core.

Example `cp_als.py tensor1000 1000,1000,1000 100,100,100 10 1,2`
The example takes a 3-mode tensor of size 1000×1000×100 with the chunk size 100×100×100 and runs the rank-10 CP ALS algorithm with materialized matricization on two modes (mode-1,2). The materialized matricization is created by cp_mat.py. The output of the example includes factor matrices (tensor1000_fac_0, tensor1000_fac_1, and tensor1000_fac_2) and a core (tensor1000_core).

1.2 `cp_mat.py`

Description Given the input tensor and mode, materialize the matricization on the modes of the tensor

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- matricized_modes (optional): modes which matricize, e.g., 1,2 - mode-1 and 2 to matricize
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include materialized matricization, *tensor_name*_0, *tensor_name*_1, etc.

Example `cp_mat.py tensor1000 1000,1000,1000 100,100,100 10 1,2`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and materialize the matricization of the tensor on two modes (mode-1,2). The output of the example includes materialized matricizations (tensor1000_0, tensor1000_1).

1.3 comp_cov_dense.py

or comp_cov_sparse.py

Description These operations compute the covariance matrices for each mode of an input tensor in dense representation (comp_cov_dense.py) or sparse representation (comp_cov_sparse.py) for Dynamic Tensor Decomposition (DTA: Dynamic Tensor Analysis), which is used to incrementally update the tensor decomposition.

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target ranks for each mode, e.g., 2,3,4
- old_tensor (optional): an old tensor on which the new tensor is incrementally updated. If it is not specified, decompose the new tensor.
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include covariance matrices for each mode of the input tensor. The covariance matrices are *tensor_name*_cov0 for mode-1, *tensor_name*_cov1 for mode-2, etc.

Example `comp_cov_*.py tensor1 1000,1000,1000 100,100,100 2,3,4`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and computes the covariance matrices of the tensor.

The output of this example includes covariance matrices (tensor1_cov0, tensor1_cov1, and tensor1_cov2).

`comp_cov_*.py tensor2 1000,1000,1000 100,100,100 2,3,4 tensor1`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and incrementally update the covariance matrices of the old tensor tensor1 with the new tensor tensor2.

The output of this example includes covariance matrices (tensor2_cov0, tensor2_cov1, and tensor2_cov2).
*: `dense`: dense representation or `sparse`: sparse representation.

1.4 comp_cov_comp.py

Description This operation computes the covariance matrices for each mode of an input tensor in dense representation using the compressed matrix multiplication (`multiply_comp` in Section 2.1.6) for Dynamic Tensor Decomposition (DTA: Dynamic Tensor Analysis), which is used to incrementally update the tensor decomposition.

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target ranks for each mode, e.g., 2,3,4
- threshold: threshold whether to run compressed matrix multiplication for computing covariance matrices. If the size of each mode of the input tensor is greater than threshold, the compressed matrix multiplication is used.

- old_tensor_name (optional): an old tensor on which the new tensor is incrementally updated. If it is not specified, decompose the new tensor.
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include covariance matrices for each mode of the input tensor. The covariance matrices are *tensor_name*_cov0 for mode-1, *tensor_name*_cov1 for mode-2, etc.

Example `comp_cov_comp.py tensor1 1000,1000,1000 100,100,100 2,3,4`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and computes the covariance matrices of the tensor.

The output of this example includes covariance matrices (tensor1_cov0, tensor1_cov1, and tensor1_cov2).

`comp_cov_comp.py tensor2 1000,1000,1000 100,100,100 2,3,4 tensor1`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and incrementally update the covariance matrices of the old tensor tensor1 with the new tensor tensor2.

The output of this example includes covariance matrices (tensor2_cov0, tensor2_cov1, and tensor2_cov2).

1.5 `dta.py`

Description Dynamic Tensor Decomposition (DTA: Dynamic Tensor Analysis). Given the input tensor, incrementally update the tensor decomposition. This operation requires the covariance matrices of the input tensor, which are computed by `comp_cov_dense.py`, `comp_cov_sparse.py`, or `comp_cov_comp.py`.

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target ranks for each mode, e.g., 2,3,4
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include factor matrices and a core. The factor matrices are *tensor_name*_fac_0, *tensor_name*_fac_1, ..., and the core is *tensor_name*_core.

Example `dtd.py tensor1 1000,1000,1000 100,100,100 2,3,4`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and decompose the tensor.

The output of this example includes factor matrices (tensor1_fac_0, tensor1_fac_1, and tensor1_fac_2) and a core (tensor1_core).

`dtd.py tensor2 1000,1000,1000 100,100,100 2,3,4 tensor1`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and incrementally update the old tensor `tensor1` with the new tensor `tensor2`.

The output of this example includes factor matrices (tensor2_fac_0, tensor2_fac_1, and tensor2_fac_2) and a core (tensor2_core).

1.6 `dta_full.py`

Description Dynamic Tensor Decomposition (DTA: Dynamic Tensor Analysis). Given the input tensor, incrementally update the tensor decomposition. This includes operations for computing the covariance matrices for an input tensor in dense representation.

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target ranks for each mode, e.g., 2,3,4
- old_tensor (optional): an old tensor on which the new tensor is incrementally updated. If it is not specified, decompose the new tensor.
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include factor matrices and a core. The factor matrices are *tensor_name*_fac_0, *tensor_name*_fac_1, ..., and the core is *tensor_name*_core.

Example `dtd_full.py tensor1 1000,1000,1000 100,100,100 2,3,4`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and decompose the tensor.

The output of this example includes factor matrices (tensor1_fac_0, tensor1_fac_1, and tensor1_fac_2) and a core (tensor1_core).

`dtd_full.py tensor2 1000,1000,1000 100,100,100 2,3,4 tensor1`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and incrementally update the old tensor `tensor1` with the new tensor `tensor2`.

The output of this example includes factor matrices (tensor2_fac_0, tensor2_fac_1, and tensor2_fac_2) and a core (tensor2_core).

Note that the initial DTA (without the old tensor to be incrementally updated) is used as the input of the Tucker decomposition.

1.7 `tucker.py`

Description Tucker Decomposition. Given the input tensor, perform the tucker decomposition. The Tucker decomposition takes HOSVD of the input tensor as the initial factor matrices and core. HOSVD of the input tensor is obtained by running DTA without an old tensor. Thus it requires to run (`dta_full.py` or `comp_cov_*.py` and `dta.py` without an old tensor.

Parameters

- tensor_name: input tensor
- tensor_size: input tensor size, e.g., 1000,1000,1000 for a tensor of size 1000×1000×1000
- chunk_size: chunk size for the input tensor, e.g., 100,100,100 for a chunk size 100×100×100
- rank: target ranks for each mode, e.g., 2,3,4
- max_iter (optional): the maximum iteration count, default value: 50
- debug (optional): 1 for debugging mode (not running the operations but showing the commands), default value: 0

Output The outputs include factor matrices and a core. The factor matrices are *tensor_name*_fac_0, *tensor_name*_fac_1, ..., and the core is *tensor_name*_core.

Example `tucker.py tensor1 1000,1000,1000 100,100,100 2,3,4`
The example takes a 3-mode tensor of size 1000×1000×1000 with the chunk size 100×100×100 and decompose the tensor. The output of the example includes 3 factor matrices (tensor1_fac_0, tensor1_fac_1, and tensor1_fac_2) and a core (tensor1_core).

2 In-Database TensorDB Operators

In this section, we introduce the novel chunk-based tensor operators (*matricization, Khatri-Rao product, Hadamard product, normalization*, and *copyArray* operators, etc.) needed for implementing in-database tensor decompositions.

In what follows, we refer to an array with two modes as `matrix` and to an array with more than two modes as `tensor`.

2.1 Chunk-based Operators

2.1.1 `matricize(tensor, m)`

Matricization transforms a tensor into a matrix along the given mode, m. More specifically, an element $(i_1, i_2, ..., i_m, ..., i_N)$ of tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times, \cdots, \times I_N}$ is mapped to (i_m, j) of mode-m matricization, $\mathbf{X}_{(m)}$, such that (assuming row-major representation of the result)

$$j = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{n=k+1 \\ n \neq m}}^{N} I_n \right) i_k,$$

Chunk-Optimized Matricization. The chunk-based mode-m matricization $\mathbf{X}_{(m)}$ of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times, \cdots, \times I_N}$ with chunks of size $K_1 \times K_2 \times, \cdots, \times K_N$ is defined as

$$\mathbf{X}_{(m)} = \left(\begin{array}{c|c|c|c} X_{11_{(m)}} & X_{12_{(m)}} & \cdots & X_{1J_{(m)}} \\ \hline X_{21_{(m)}} & X_{22_{(m)}} & \cdots & X_{2J_{(m)}} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline X_{I_m 1_{(m)}} & X_{I_m 2_{(m)}} & \cdots & X_{I_m J_{(m)}} \end{array} \right),$$

where $X_{ij_{(m)}}$ is the (i,j)-th chunk of $\mathbf{X}_{(m)}$ and $J = I_1 \times \cdots \times I_{m-1} \times I_{m+1} \times \cdots \times I_N$.

An element of $(i_1, i_2, \ldots, i_m, \ldots, i_N)$ in a chunk of $(c_1, c_2, \ldots, c_m, \ldots, c_N)$ of $\mathcal{X}$ is mapped to an element of (i_m, j) in a chunk of (c_m, d) of $\mathbf{X}_{(m)}$, such that

$$j = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{l=k+1 \\ l \neq m}}^{N} K_l \right) i_k \text{ and } d = \sum_{\substack{k=1 \\ k \neq m}}^{N} \left(\prod_{\substack{l=k+1 \\ l \neq m}}^{N} \lceil I_l / K_l \rceil \right) c_k. \quad (1)$$

Materialization of the Results of the Matricization Operation. Tensor matricization is a costly operation, requiring at least one full read-and-write of the tensor data. Moreover, in CP decomposition, matricization of all modes of tensor are needed in each iteration. Therefore, one way to minimize the overall matricization overhead, is to *materialize the matricization results* (see Section 1.2). More specifically, once a matricization is computed, the result can be materialized on disk and this materialized matricization can be is used in all subsequent iterations.

2.1.2 `khatrirao(left_matrix, right_matrix)`

Given a left matrix, $\mathbf{A} \in \mathbb{R}^{I \times K}$, and a right matrix, $\mathbf{B} \in \mathbb{R}^{J \times K}$, their Khatri-Rao product is denoted by $\mathbf{A} \odot \mathbf{B}$. The result is a matrix of size $(IJ) \times K$, defined as

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1 \; \mathbf{a}_2 \otimes \mathbf{b}_2 \cdots \mathbf{a}_n \otimes \mathbf{b}_n \cdots \mathbf{a}_K \otimes \mathbf{b}_K],$$

where $\mathbf{a}_n$ and $\mathbf{b}_n$ are columns of $\mathbf{A}$ and $\mathbf{B}$, respectively and $\otimes$ is the Kronecker product. Note that the Kronecker product, $\mathbf{U} \otimes \mathbf{V}$, of matrices $\mathbf{U} \in \mathbb{R}^{x \times y}$ and $\mathbf{V} \in \mathbb{R}^{w \times z}$ results in matrix of size $(xw) \times (yz)$, where

$$\mathbf{U} \otimes \mathbf{V} = \begin{pmatrix} u_{11}\mathbf{V} & u_{12}\mathbf{V} & \cdots & u_{1y}\mathbf{V} \\ u_{21}\mathbf{V} & u_{22}\mathbf{V} & \cdots & u_{2y}\mathbf{V} \\ \vdots & \vdots & \ddots & \vdots \\ u_{x1}\mathbf{V} & u_{x2}\mathbf{V} & \cdots & u_{xy}\mathbf{V} \end{pmatrix}.$$

We define the chunk-based Khatri-Rao product for matrices $\mathbf{A}$ (with chunks $A_{11}, ..., A_{IJ}$) and $\mathbf{B}$ (with chunks $B_{11}, ..., B_{IJ}$), as follows:

$$\mathbf{A} \odot \mathbf{B} = \left(\begin{array}{c|c|c|c} A_{11} \odot B_{11} & A_{12} \odot B_{12} & \cdots & A_{1J} \odot B_{1J} \\ \hline A_{21} \odot B_{21} & A_{22} \odot B_{22} & \cdots & A_{2J} \odot B_{2J} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline A_{I1} \odot B_{I1} & A_{I2} \odot B_{I2} & \cdots & A_{IJ} \odot B_{IJ} \end{array}\right).$$

2.1.3 `Hadamard(left_matrix, right_matrix)`

The Hadamard product is the elementwise matrix product; more specifically, given matrices $\mathbf{A}$ and $\mathbf{B}$, both of size $I \times J$, their Hadamard product, denoted as $\mathbf{A} * \mathbf{B}$, results in the following size $I \times J$ matrix:

$$\mathbf{A} * \mathbf{B} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{pmatrix}.$$

Given this, we define the chunk-based Hadamard product for matrices $\mathbf{A}$ with chunks $A_{11}, ..., A_{IJ}$ and $\mathbf{B}$ with chunks $B_{11}, ..., B_{IJ}$ as follows:

$$\mathbf{A} * \mathbf{B} = \left(\begin{array}{c|c|c|c} A_{11} * B_{11} & A_{12} * B_{12} & \cdots & A_{1J} * B_{1J} \\ \hline A_{21} * B_{21} & A_{22} * B_{22} & \cdots & A_{2J} * B_{2J} \\ \hline \vdots & \vdots & \vdots & \vdots \\ \hline A_{I1} * B_{I1} & A_{I2} * B_{I2} & \cdots & A_{IJ} * B_{IJ} \end{array}\right).$$

Again, the data movement is constrained within individual chunks and, thus, the order in which chunks are considered does not impact performance.

2.1.4 `copyArray(operator(args), array)`

This operator copies the result of `operator(args)` to a temporary array, `array`, and is used for updating the intermediate results (e.g., in the in-database CP decomposition, for updating the factor matrices, which get updated in each iteration). In contrast, SciDB's analogous operation, `store`, does not update an existing array but creates a new version of the array (also maintaining the previous versions). Also, unlike `store`, `copyArray` does not use run-length encoding/decoding, since frequently updated and relatively small factor matrices, do not benefit from run-length encoding/decoding. Note that the `copyArray` operator is used to update a temporary array in the memory, thus in order to update the array permanently in the database, use the `store` operator of SciDB.

2.1.5 `multiply_row(matrix1, matrix2)`

Matrix multiplication of entries of *rows* of each matrix so that both matrices can take advantage of the row-wise data ordering. The two matrices must have the same size of 'row' dimension and same chunk size along that dimension. E.g., $n \times m$ matrix *times* $n \times m$ matrix generates a result matrix of $m \times m$.

2.1.6 `multiply_comp(matrix1, matrix2)`

Approximate matrix multiplication of entries of *columns* of each matrix using compressed matrix multiplication. For compressed matrix multiplication, columns by columns can be more efficient than rows by rows since the inner for loop in compressed matrix multiplication iterates on entries of each column of the matrices. E.g., $n \times m$ matrix *times* $n \times m$ matrix generates a result matrix of $n \times n$. Note that the compressed matrix multiplication is only used on dense matrices. If the input matrix is sparse enough, the sparse matrix multiplication is used.

2.2 Other Chunk-based Operators

In addition to the above, our in-database CP implementation also requires chunk-based implementations of other operators as followings. These operators are used for CP ALS algorithm (`cp_als.py`).

- `twoNormArray(matrix)` - Normalize the columns of the input `matrix` by 2-norm
- `twoNormCoreArray(matrix, core)` - Normalize the columns of the input `matrix` by 2-norm and the weight is copied into core
- `maxNormArray(matrix)` - Normalize the columns of the input `matrix` by max-norm
- `maxNormCoreArray(matrix)` - Normalize the columns of the input `matrix` by max-norm and the weight is copied into core
- `norm_residual(tensor1, tensor2)` - Frobenius norm of the difference between the given `tensor1` and `tensor2` used for fit computation.

2.3 Non-Chunked Tensor Operations

In addition to the above chunked operators, we also implement two non-chunked operators, *pseudoinverse* and *eigen-decomposition*. While these require their inputs to fit into the memory, since (during tensor decomposition) inputs are often relatively small matrices, this rarely constitutes a problem.

2.3.1 `pinverse(matrix)`

This operator returns the pseudo-inverse of the input `matrix`. We implement this operator using a C++ linear algebra library from [5], where SVD is used to solve pseudo-inverse problem. Since during CP decomposition, the input to the pseudo-inverse operation is a matrix of size `rank` $\times$ `rank`, where `rank` is a relatively small number of target components, this matrix easily fits the main memory and does not require a chunk-based implementation.

2.3.2 `eigen(matrix, r)`

This operator returns `r` leading eigen-vectors of the input `matrix`. Similar to the `pseudoinverse` operator, eigen-decomposition is an in-memory operation and we use the eigen-decomposition function provided in [5] for implementation.

We use this eigen-decomposition operation to implement incremental tensor decomposition. In particular, we take the leading eigen-vectors of the $I_d \times I_d$ covariance matrix to generate factor matrices, where I_d is the size of the mode d of the tensor. Note that this matrix is often much smaller than the whole tensor and, thus, we assume that the covariance matrix fits into the main-memory.

References

[1] `http://www.scidb.org/`

[2] J. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition. Psychometrika, 1970.

[3] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an explanatory multi-modal factor analysis. UCLA Working Papers in Phonetics, 16(1):84, 1970.

[4] M. Kim (2014). TensorDB and Tensor-based Relational Model (TRM) for Efficient Tensor-Relational Operations. Ph.D. Thesis. Arizona State University.

[5] C. Sanderson et al. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.

[6] Sun et al. Incremental tensor analysis: Theory and applications. ACM Trans. Knowl. Discov. Data, 2(3):11:111:37, 2008.

[7] L. Tucker. Some mathematical notes on three-mode factor analysis. Psychometrika, 31(3):279311, Sept. 1966.