

Finding a Second Wind: Speeding Up Graph Traversal Queries in RDBMSs Using Column-Oriented Processing

Mikhail Firsov¹[0000-0001-6739-7303], Michael Polyntsov¹[0000-0001-7356-2504],
Kirill Smirnov¹[0000-0003-4727-3455], and George
Chernishev¹[0000-0002-4265-9642]

Saint-Petersburg University, Russia

{mikhail.a.firsov, polyntsov.m, kirill.k.smirnov, chernishev}@gmail.com

Abstract. Recursive queries and recursive derived tables constitute an important part of the SQL standard. Their efficient processing is important for many real-life applications that rely on graph or hierarchy traversal. Position-enabled column-stores offer a novel opportunity to improve run times for this type of queries. Such systems allow the engine to explicitly use data positions (row ids) inside its core and thus, enable novel efficient implementations of query plan operators.

In this paper, we present an approach that significantly speeds up recursive query processing inside RDBMSes. Its core idea is to employ a particular aspect of column-store technology (late materialization) which enables the query engine to manipulate data positions during query execution. Based on it, we propose two sets of Volcano-style operators intended to process different query cases.

In order validate our ideas, we have implemented the proposed approach in PosDB, an RDBMS column-store with SQL support. We experimentally demonstrate the viability of our approach by providing a comparison with PostgreSQL. Experiments show that for breadth-first search: 1) our position-based approach yields up to 6x better results than PostgreSQL, 2) our tuple-based one results in only 3x improvement when using a special rewriting technique, but it can work in a larger number of cases, and 3) both approaches can't be emulated in row-stores efficiently.

Keywords: Query Processing · Column-stores · Recursive Queries · Late Materialization · Breadth-First Search · PosDB.

1 Introduction

The ANSI'99 SQL standard introduced the concept of recursion into SQL with syntactic constructs to define recursive views and recursive derived tables. This allows users to store graph data in a tabular form and to express some graph queries using CTEs and recursive syntax. The admissible subset is rather limited compared to the specialized graph systems, but it is sufficient to solve a number of common tasks. Such tasks originate from many real-life applications and

usually concern some hierarchy traversal which comes as a breadth-first search computation.

In this paper, we present another outlook on RDBMS architecture that significantly improves system performance at least for some types of graph queries expressed by recursive SQL. More specifically, we present a column-oriented approach that will improve run times for queries that perform breadth-first search.

Having emerged about fifteen years ago, column-stores quickly became ubiquitous in analytic processing of relational data. Their idea is simple: store data in a columnar form in order to read only necessary columns for evaluating the query. Such approach also provides better data compression rates [1], improves CPU cache utilization, facilitates SIMD-enabled data processing, and offers other benefits. However, some column-stores additionally allow the query engine to explicitly use data positions during query execution. This made way for a number of optimizations and techniques that offered various benefits for query processing. Thus, we differentiate the “position-enabled” column-stores from the rest as the column-stores that are able to reap benefits from explicit position manipulation inside their engine. We believe that such an approach can give RDBMs a second wind in handling graph queries.

Positions (also called row ids or offsets) are integers that refer to some record or individual attribute value inside a table. Operating on data positions allows query engine to achieve savings by deferring switching to data values. This group of techniques is called late materialization and it was successfully employed for various query plan operators [18,2,20,12].

We employ this technique to design two sets of Volcano-style [7] operators intended to handle different query cases that involve recursive processing. We have implemented them inside a position-enabled column-store PosDB [4,3]. Next, in order to evaluate them we run experiments with queries that perform breadth first search. We have also performed the comparison of our approach with PostgreSQL.

The overall contribution of this paper is the following:

1. A survey of existing query processing techniques in RDBMSs that concern recursive queries.
2. A design of two query operators for position-enabled column store that speed up recursive query evaluation.
3. An experimental evaluation of proposed techniques and a comparison with state-of-the-art row-store RDBMS.

This paper is organized as follows. In Section 2 we survey various aspects of implementation and usage of recursive queries inside relational DBMSs. Then, in Section 3 we present the main features of PosDB and discuss its query processing internals. After this, in Section 4 we describe implementation details of the proposed recursive operators and their use in the existing query plan model of PosDB. Section 5 contains an evaluation that compares PosDB with PostgreSQL using a series of experiments on trees of different size, height and additional payload. Finally, in Section 6 we conclude this paper and discuss future work.

2 Related Work and Motivation

2.1 Related Work

In this section we review existing papers that address graph query processing in SQL-supporting systems, paying special attention to recursive evaluation.

One of the earliest papers that addressed the problem of recursive query evaluation was the paper [11]. There, author introduces several query optimizations for recursive queries with graphs, namely: early evaluation of row selection conditions, elimination of duplicate rows in intermediate tables, and using an index to accelerate join computation.

The authors of the papers [6,5] describe several issues with query optimization in relational databases when implementing recursive queries. The first approach they mention is the full feedback approach (FFB), which provides the optimizer with the demographics of each recursion iteration so that it can generate a new plan for the subsequent iteration. However, FFB interrupts potential pipelining and cannot take advantage of global query optimizations, making it unsuitable for parallel DBMS. The next approach, look ahead with feedback (LAWF), generates plans for the subsequent k iterations in advance, with k depending on the query planning cost and propagation of join estimation errors. The authors present a dynamic feedback mechanism based on passive monitoring to collect feedback and to determine when re-planning is necessary. The LAWF method supports both pipelining and global query optimization.

In the paper [14] the authors consider two graph problems: transitive closure computation and adjacency matrix multiplication. In order to solve them, the authors study the optimization of queries that involve recursive joins and recursive aggregations in column- and row-oriented DBMS. They evaluate the impact of several query optimization techniques, such as pushing aggregation through recursion and using ORDER BY with merge joins in column-store instead of hash joins. The authors evaluate effects of indexing columns containing vertices and effects of sorting rows in a row-store to evaluate the iteration of k joins.

In the paper [13] the author evaluates various recursive query optimizations for the plan generator. The paper considers five techniques: storage and indexing for efficient join computation, early selection, early evaluation of non-recursive joins, pushing duplicate row elimination, and pushing aggregation. The author uses four types of graphs: tree, list, cyclic, and complete graphs. However, similarly to previous work [14] author uses a sequence of SQL commands (including INSERTs) to implement the proposed optimizations. Such approach may suffer from various overheads, as opposed to implementing an operator node in the engine source code. This, in turn, may lead to inaccurate results.

The Recursive-aggregate-SQL (RaSQL) [8] system extends the current SQL standard to support aggregates in recursion. It can express powerful queries and declarative algorithms, such as graph and data mining algorithms. The RaSQL compiler allows mapping declarative queries into one basic fixpoint operator supporting aggregates in recursive queries. The aggregate-in-recursion optimization

brought by the PreM property and other improvements make the RaSQL system more performant than other similar systems.

In the paper [17] the authors address the problem of storing large property data graphs inside relational DBMS. They adapt the SQLGraph [19] approach to reduce the disk volume and increase processing speeds. They evaluate their schema using the PostgreSQL on LDBC-SNB and show that their schema not only performs better on read-only queries but also performs better on workloads that include update operations.

Graph databases are good for storing and querying provenance data. One of the earliest papers that evaluated this possibility was the study [23]. The authors compare relational and graph databases on different types of queries. This study demonstrated that for traversal queries, graph databases were clearly faster, sometimes by a factor of 10. This result was expected since relational databases are not designed to perform traversals such as standard breadth-first-search.

Another paper that concerned graph databases in data provenance domain was the study [16]. The authors propose an improved version of the DPHQ framework for capturing and querying the provenance data. They conclude that graph databases offer significant performance gains over relational databases for executing multi-depth queries on provenance. The performance gains become much more pronounced with the increase in traversal depth and data volumes.

2.2 Motivation

The related works discussed above demonstrated popularity and relevance of graph queries and graph database systems. However, they also showed that there is only a handful of studies that address processing of graph queries (BFS, transitive closure) using the recursion technique in SQL-supporting systems.

Moreover, despite the existence of studies which touch upon the processing of recursive SQL queries in column-stores (e.g. [14]), there are no studies that propose to leverage data positions. On the other hand, in our paper we propose an in-depth operator redesign, which is based on this idea.

3 Background

PosDB is a disk-based distributed column-store which features explicit position manipulation, i.e. it is a “position-enabled” system. In this regard it is close to the ideas of early systems such as the C-Store [18] and the MonetDB [10].

PosDB uses the pull-based Volcano model [7] with block-oriented processing. Its core idea is to employ two types of intermediate representations: tuple- and position-based. In the tuple-based representation operators exchange blocks of value tuples. This type of representation is similar to most existing DBMSs. On the other hand, position-based representation is a characteristic feature of PosDB. In the positional form, intermediates are represented by a generalized join index [22] which is presented in Fig. 1a. Join index stores an array of record indices, i.e. positions, for each table it covers (top of Fig. 1a). Tuples are encoded

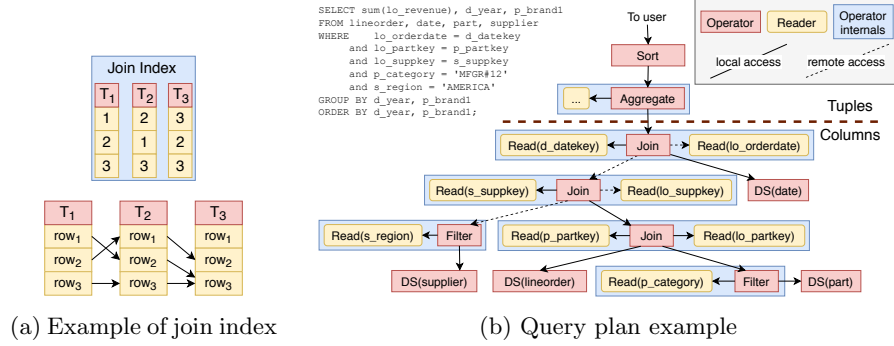


Fig. 1: PosDB internals

using rows in the join index. Most operators in PosDB are either positional or tuple-based, with positional ones having specialized Reader entities for reading values of individual table attributes. The query plan in PosDB is divided into positional and tuple parts, and the moment of converting positions into tuples is called materialization. Materialization is to be performed at some moment of query plan, since user needs not positions, but tuples. It can be performed by either a special **Materialize** operator or by some operators, such as an aggregation operator.

In the query plan presented in Fig. 1b, the materialization point is indicated by a brown dotted line. Below the line, positional representation is used and above the line uses tuple representation. In the latest version of PosDB, a query plan may contain several materialization points, in such a manner that every leaf-root path will have one.

Such architecture leads to several different classes of query plans which are discussed in reference [4]. Operating on positions instead of tuple values allows to achieve significant cost savings for some queries. For example, in case of filtering join it is possible to reduce the total amount of data read from disk if join will be performed on positions first, and then the rest of necessary columns will be read. This is a general idea of late materialization and it was extensively used for implementing many [1,9,12,21,15] operators and their combinations. At the same time, in PosDB it is possible to build plans equivalent to naive [1] column-stores, i.e. which will read only necessary columns, construct tuples and continue as it was row-store. In this paper we are going to discuss an application of this technique for processing recursive queries.

PosDB is a large project and it has many features and implementation details. However, they are out of the scope of this work and are not necessary for its understanding. A detailed description of baseline architecture can be found in paper [3], and the recent additions are described in [4]. Finally, an interactive demo of PosDB can be seen at the following link¹.

¹ <https://pos-db.com/>

4 Proposed Approach

In order to implement recursive queries in the PosDB, we have introduced two new operator groups into its operator set. These groups share the same use pattern and differ only in the used data representation (rows or positions). Their generalized operation flow is presented in Fig. 2, which is as follows:

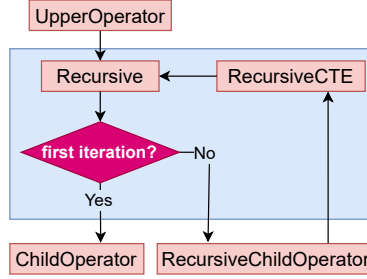


Fig. 2: Sample query plan representation using Recursive and RecursiveCTE

- The **Recursive** operator stores pointers to **RecursiveCTE**, **ChildOperator** and **RecursiveChildOperator**. **ChildOperator** is used for the non-recursive part of the query, with its help PosDB gets initial rows or initial positions. The **RecursiveChildOperator** is a regular operator, but internally it either explicitly or implicitly (via several intermediate operators) receives data from **RecursiveCTE**.
- **RecursiveCTE** stores a pointer to the **Recursive**, from which it asks for new records to be passed by the **Next** method in **RecursiveChildOperator**.

Recall that there are two types of intermediate data representation in PosDB: tuple-based and positional. This results in two sets of operators:

- **TRecursive** and **TRecursiveCTE** that only work with blocks of tuples.
- **PRRecursive** and **PRRecursiveCTE** that only work with position blocks.

We have designed only these two sets, each focusing on one particular data representation, either tuple-based or positional. However, the first thing which comes to mind is to use a combination of tuple-based and positional operators. For example, consider a case when **ChildOperator** and **RecursiveCTE** return a position block and **RecursiveChildOperator** returns a tuple block. In this case, the query engine will have to translate the tuples received from **RecursiveChildOperator** back to positions in order to use them for the second and subsequent steps of the recursion. However, this may be impossible in certain circumstances, if, for example, a generated attribute (e.g. $value * 2 + 1$) is present in the tuple block. In this case, there will be no original column which may be pointed to by a position.

For the sake of clarity, we are going to work through an example. Consider the following recursive query to find all neighbors of a vertex with $\text{id} = 0$ up to depth 4:

Listing 1.1: Recursive query example

```

1 WITH RECURSIVE edges_cte (id, from, to) AS
2   (SELECT edges.id, edges.from, edges.to
3    FROM edges WHERE edges.from = 0
4    UNION ALL
5     SELECT edges.id, edges.from, edges.to
6     FROM edges JOIN edges_cte AS e
7     ON edges.from = e.to_v)
8 SELECT edges_cte.id, edges_cte.from, edges_cte.to
9 FROM edges_cte
10 OPTION (MAXRECURSION 4);

```

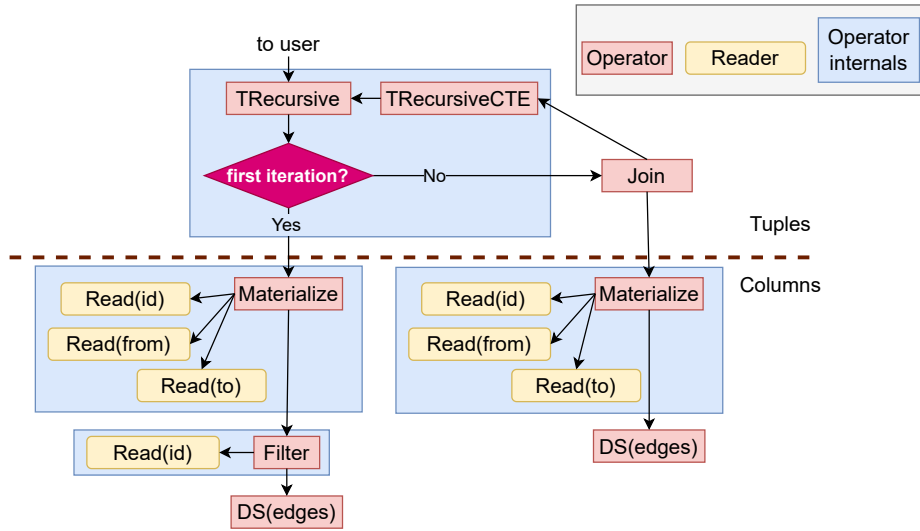


Fig. 3: Query tree using TRecursive and TRecursiveCTE

The plan of this query using the introduced structures can be represented via the diagram in Fig. 3. In this figure:

- The left **Materialize** operator is a **ChildOperator**: it will be executed once in order to initialize the starting set of tuples.
- The **Join** is a **RecursiveChildOperator**.
- The set of tuple blocks of the current recursion step is stored inside **TRecursive**: we will call it **curLevel**. In addition, **TRecursive** will store the position of the block in **curLevel**, which should be passed to **TRecursive** next time.

The evaluation itself is as follows:

1. **TRecursive** requests blocks from the left **Materialize** as long as they are not empty and stores them in **curLevel**.
2. **TRecursive** passes all blocks from **curLevel** up until it reaches the end of **curLevel**.
3. To get the block of the new recursion step, **TRecursive** requests the block from **Join**.
4. **Join** requests blocks from **TRecursiveCTE** and from the right **Materialize**.
5. **TRecursiveCTE** asks **TRecursive** for blocks.
6. **TRecursive** increments the internal counter and passes **TRecursiveCTE** in response to its requests for blocks from **curLevel**.
7. After typing a new block of a certain size, **Join** gives it to **TRecursive**. If it is a non-empty block, then **TRecursive** will store it in **nextLevel** temporary storage and proceed to Step 3. If it is an empty block, then **curLevel** is replaced with **nextLevel**. If now **curLevel** is an empty set of blocks, then we say that **TRecursive** has finished its work, otherwise **TRecursive** proceeds to Step 2.

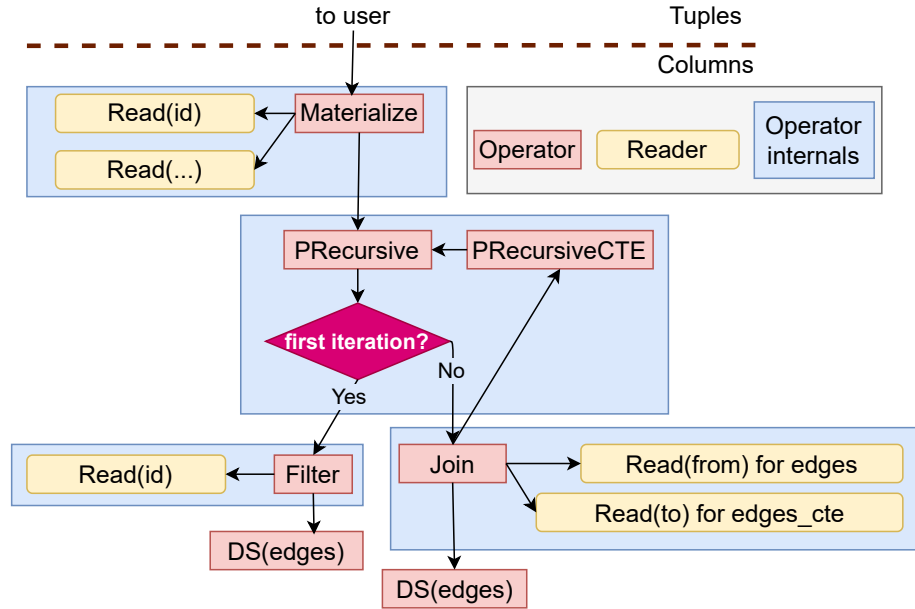


Fig. 4: Query tree using **PRRecursive** and **PRRecursiveCTE**

The plan of the same query using **PRRecursive** operator can be represented via the diagram in Fig. 4. Evaluation of this query tree will proceed in a similar

way. An important limitation here is that we can only work with positions of the same table, which means that the **Join** operator must return the positions of the same table as the left **Filter** operator. However, **curLevel** stores not tuple blocks, but positions blocks. In all other respects, the logic of the **PRRecursive** and **PRRecursiveCTE** operators is completely identical to the logic of their tuple-based counterparts.

5 Evaluation

5.1 Methodology

We evaluate our implementations using hierarchical recursive queries on a tree graph. We generated the datasets for the experiments with a simple script². All evaluated queries solve the task of finding all nodes that lie at a distance of n hops from the root using the BFS algorithm. A test graph was stored in PosDB and PostgreSQL as an edge list. Columns are of the following types: **id**, **from**, **to** are **int** (4 bytes); **name** is **varchar(15)** (32 bytes); each additional column in the second and third test sets is **varchar(20)** (42 bytes). The number of table rows is indicated above the figures with test results.

In order to evaluate our solution, we have selected a baseline of PostgreSQL. Our choice is based on the following considerations. First of all, we needed a classic row-store system in order to demonstrate the advantages of our approach. Second, PostgreSQL meets another important requirement: it is free from the DeWitt clause³.

PostgreSQL was configured as follows: JIT compilation and parallelism were disabled since JIT compilation is not implemented in PosDB and enabling parallelism would add unnecessary complexity without contributing anything important in the scope of this paper. To ensure that hash join is used in the query plan as in PosDB, merge and nested loop joins were turned off using planner parameters. The **temp_buffers** and **work_mem** parameters were set to values that ensure that any table under test fits into memory. To prevent caching from affecting the results, PostgreSQL internal caches were cleared between runs.

PosDB buffer manager was set to 1GB (32K pages of 32KB size).

Each experiment was repeated 10 times, and the average of the results was calculated.

We pose the following research questions:

- RQ1 Does our position-based approach bring any performance gain in a special case when all attributes of a table are required in the recursive part of a query?
- RQ2 What performance advantage does our position-based approach offer when introducing additional payload through auxiliary attributes used in projection?

² https://github.com/Firsov62121/tree_generator

³ <https://www.brentozar.com/archive/2018/05/the-dewitt-clause-why-you-rarely-see-database-benchmarks/>

RQ3 Is it possible to emulate our approach inside a row-store by rewriting a query in such a way that it will keep a minimum number of columns inside the recursive core and then join the rest?

To answer these questions, we have designed the following experiments, corresponding to each RQ:

1. For the first experiment, we used a BFS query with the table consisting only of attributes required for the traversal, giving no benefits to PosDB (see Listing 1.1 and corresponding plans in Figures 3,4).
2. For the second experiment, we used the query from the first experiment modified by adding payload attributes to the input table itself and to all projections. SQL queries of the following type were used for PostgreSQL and PosDB:

```

1 WITH RECURSIVE edges_cte (id, from, to, column1,
2   ..., columnN, depth) AS
3   (SELECT edges.id, edges.from, edges.to,
4     edges.column1, ..., edges.columnN, 0
5    FROM edges WHERE edges.id = 0
6    UNION ALL
7     SELECT edges.id, edges.from, edges.to,
8       edges.column1, ..., edges.columnN,
9       e.depth + 1 FROM edges JOIN edges_cte AS e
10      ON edges.from = e.to AND e.depth < DEPTH_VAL)
11 SELECT edges_cte.id, edges_cte.from, edges_cte.to,
12   edges_cte.column1, ..., edges_cte.columnN
13 FROM edges_cte;
```

3. For the third experiment, we have created a special type of query:

```

1 WITH RECURSIVE edges_cte(id, to, depth) AS
2   (SELECT edges.id, edges.to, 0 FROM edges
3    WHERE edges.from = 0
4    UNION ALL
5     SELECT edges.id, edges.to, e.depth + 1
6    FROM edges JOIN edges_cte AS e
7    ON edges.from = e.to AND e.depth < DEPTH_VAL)
8 SELECT edges.id, edges.to, edges.from,
9   column1, ..., columnN FROM edges JOIN
10  edges_cte ON edges.id = edges_cte.id;
```

In all plans of all evaluated systems the `edges_cte` was hashed in the hash join (default PostgreSQL behavior).

5.2 Experimental Setup

Experiments were performed using the following hardware and software configuration. Hardware: Intel® Core™ i7-8550U CPU @ 1.80GHz (8 cores), 16 GiB RAM, 500GB Samsung PSSD T7. Software: Ubuntu 20.04.5 LTS x86_64, Kernel 5.15.0-60-generic, gcc 9.4.0, PostgreSQL 14.2.

5.3 Experiments & Discussion

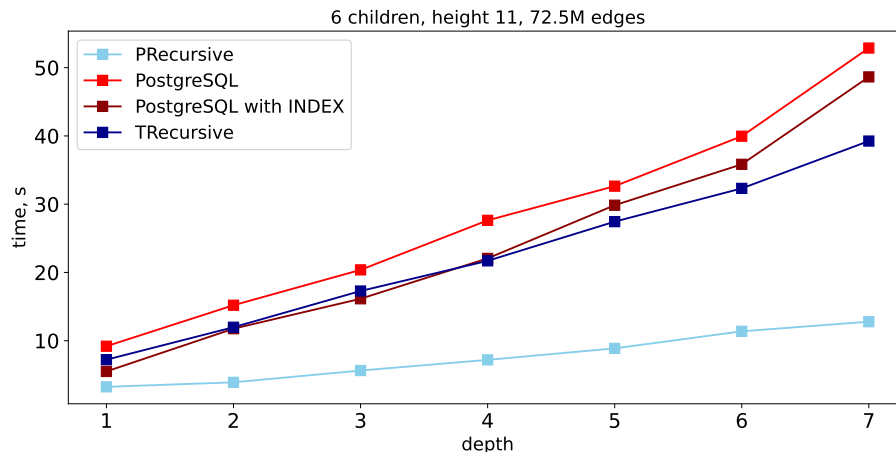


Fig. 5: Experiment 1 results

Experiment 1. The results are presented in Figure 5. The **TRecursive** approach exhibits performance that is similar to that of PostgreSQL, as expected, since the underlying query engines perform identical operations. Meanwhile, **PRecursive** outperforms **TRecursive** significantly, because it uses only two out of the four attributes (**from** and **to**) during the search, and materializes values of the third attribute (**id**) only when the required table rows are known. The number of rows scheduled to be materialized is much smaller than the total number of rows in the table (by roughly 200 times), resulting in operators passing around intermediate results of much smaller size. The index in PostgreSQL was built over **from**, since it is used to find edges in the join in the recursive part of the query. As we can see, this helps improve PostgreSQL performance, although it is a small improvement. Moreover, with the increase in depth, **TRecursive** demonstrates slightly better results compared to PostgreSQL with Index.

Experiment 2. The queries considered in this experiment were executed on a dataset with an additional parameter, denoted as N , which corresponds to the number of additional columns. The query plans for PosDB and PostgreSQL remain almost identical to those used in the first experiment, with the addition of ancillary columns to the **Materialize** operators in PosDB and to the projections in PostgreSQL, respectively.

Due to space constraints, we did not include **PostgreSQL with INDEX** in this and subsequent experiments, as its behavior is equivalent to that of PostgreSQL when changing the parameter N . Furthermore, results of **PRecursive** were only included for the maximum number of additional columns, as the time taken by **PRecursive** was predictably found to be almost independent of N .

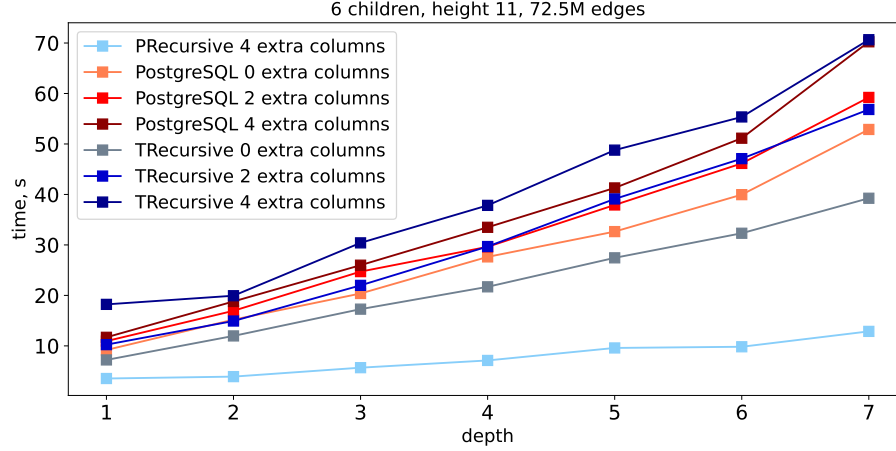


Fig. 6: Experiment 2 results

The run times of the queries depending on the depth of the traversal are presented in Fig. 6. As we can see, with increasing depth, the gap in the runtime on tables of different “widths” grows. PosDB **PRRecursive** outperforms all other approaches. This happens due to late materialization reducing sizes of intermediate results significantly. In this experiment, it matters even more due to the substantial overhead associated with passing “wide” intermediates (all columns) between operators, even though only two of them are required for the recursive part (**from** and **to**). It’s important to mention that as the “width” of passed row grows, PosDB **TRecursive** falls behind PostgreSQL. This happens because of columnar nature of PosDB. With **TRecursive** it requires more disk accesses (one for each column) to retrieve a single row from a table. In contrast, PostgreSQL can do this with a single access since all the data for table rows is stored together.

Experiment 3. This experiment is similar to the previous one, but now we are trying to conserve space in **edges_cte** by reducing the size of the intermediates. We only store the data necessary for reconstructing the original table row and navigating through the tree. This query requires more RAM since a second copy of the edges table is required by the top-level join. Therefore, we had to reduce the dataset due to the memory constraints of the employed hardware.

The run times of the queries depending on the depth of the traversal are presented in Fig. 7. It can be seen that the performance of **PRRecursive** is similar to its performance in the previous experiment, it has the best performance in this experiment too among all compared approaches. **TRecursive**, however, beats PostgreSQL in this experiment.

As we can see by the performance improvement of **TRecursive**, this method helps reduce disk access overhead of row reconstruction inside the **TRecursive** operator. In PosDB, unnecessary columns are now only materialized once at

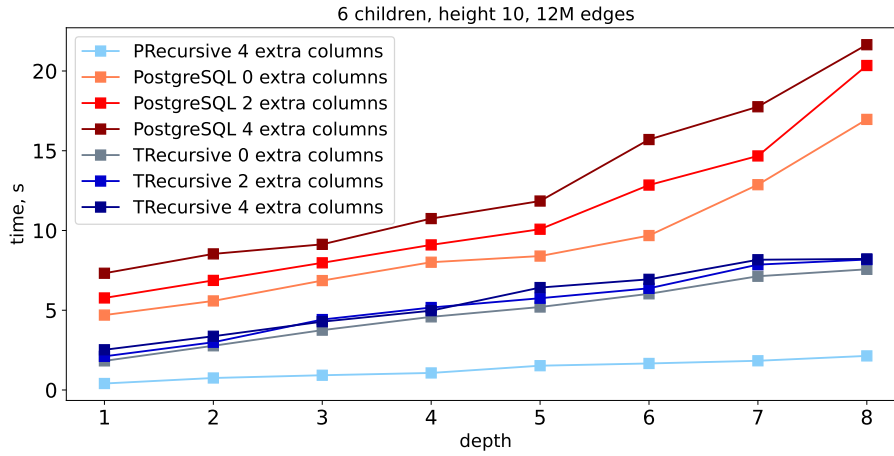


Fig. 7: Experiment 3 results

the very end. Whereas in **PostgreSQL**, the internal hash join involves inefficient sequential data reads that discard unnecessary columns. Finally, this experiment shows that our approach cannot be emulated inside **PostgreSQL** via join.

6 Conclusion

In this paper, we proposed two approaches to implementing recursive queries in a position-enabled column-oriented DBMS: **TRecursive** and **PRRecursive**, with the latter utilizing positions to implement the late materialization approach. We implemented two sets of operators: 1) a tuple-based set which is similar to the operators that can be found in classical row-stores but leveraging columnar data access, and 2) a positional-based set which is the main contribution of the paper.

We conducted experiments to evaluate the performance of the proposed approaches and used **PostgreSQL** as the baseline. Experiments demonstrated that both approaches offer improvement, with **PRRecursive** offering up to 6 times performance gain over **PostgreSQL** and 3 times over **TRecursive**. However, **TRecursive** remains the only option if there are two (or more) distinct tables in the **RECURSIVE** part used, due to implementation-related restrictions. Also, **TRecursive** yields up to 3x improvement over **PostgreSQL** when an additional payload columns which are not required in the **RECURSIVE** part exist and the query can be rewritten to project them only in the top-level. Finally, we shown that it is not possible to emulate our approach inside a row-store efficiently.

References

1. Abadi, D., Boncz, P., Harizopoulos, S.: The Design and Implementation of Modern Column-Oriented Database Systems. Now Publishers Inc. (2013)

2. Boncz, P.A., Kersten, M.L.: Mil primitives for querying a fragmented world. *The VLDB Journal* **8**(2), 101–119 (Oct 1999)
3. Chernishev, G.A., Galaktionov, V.A., Grigorev, V.D., Klyuchikov, E.S., Smirnov, K.K.: PosDB: An Architecture Overview. *Programming and Computer Software* **44**(1), 62–74 (Jan 2018)
4. Chernishev, G.A., Galaktionov, V., Grigorev, V.V., Klyuchikov, E., Smirnov, K.: A comprehensive study of late materialization strategies for a disk-based column-store. In: *DOLAP@EDBT’22. CEUR*, vol. 3130, pp. 21–30 (2022)
5. Ghazal, A., Crolotte, A., Seid, D.: Recursive sql query optimization with k-iteration lookahead. In: *DEXA’06*. pp. 348–357. Springer Berlin Heidelberg (2006)
6. Ghazal, A., Seid, D., Crolotte, A., Al-Kateb, M.: Adaptive optimizations of recursive queries in teradata. *SIGMOD’12* p. 851–860 (2012)
7. Graefe, G.: Query evaluation techniques for large databases. *ACM Comput. Surv.* **25**(2), 73–169 (Jun 1993)
8. Gu, J., et al.: RaSQL: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. *SIGMOD’19* p. 467–484 (2019)
9. Harizopoulos, S., Abadi, D., Boncz, P.: Column-oriented database systems, *vldb 2009 tutorial*. (2009), nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf
10. Idreos, S., et al.: MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* **35**(1), 40–45 (2012)
11. Jachiet, L., Genevès, P., Gesbert, N., Layaïda, N.: On the optimization of recursive relational queries: Application to graph queries. *SIGMOD’20* p. 681–697 (2020)
12. Mukhaleva, N., Grigorev, V., Chernishev, G.: Implementing window functions in a column-store with late materialization. In: *MEDI’19*. pp. 303–313 (2019)
13. Ordóñez, C.: Optimization of linear recursive queries in sql. *IEEE Transactions on Knowledge and Data Engineering* **22**(2), 264–277 (2010)
14. Ordóñez, C., Gurram, A., Rai, N.: Recursive query evaluation in a column dbms to analyze large graphs. *DOLAP’14* p. 71–80 (2014)
15. Polyntsov, M., Grigorev, V., Smirnov, K., Chernishev, G.: Implementing the comparison-based external sort. In: *New Trends in Database and Information Systems*. pp. 500–511. Springer International Publishing, Cham (2022)
16. Rani, A., Goyal, N., Gadia, S.K.: Efficient multi-depth querying on provenance of relational queries using graph database. *Proceedings of the 9th Annual ACM India Conference* p. 11–20 (2016)
17. Schmid, M.: On efficiently storing huge property graphs in relational database management systems. *iiWAS’19* p. 344–352 (2019)
18. Stonebraker, M., et al.: C-Store: A Column-Oriented DBMS. p. 553–564. *VLDB ’05, VLDB Endowment* (2005)
19. Sun, W., et al.: Sqlgraph: An efficient relational-based property graph store. p. 1887–1901. *SIGMOD ’15* (2015)
20. Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query Processing Techniques for Solid State Drives. pp. 59–72. *SIGMOD ’09* (2009)
21. Tuchina, A., Grigorev, V., Chernishev, G.: On-the-fly filtering of aggregation results in column-stores. In: *SEIM’18*. pp. 53–60. No. 2135 in *CEUR* (2018)
22. Valduriez, P.: Join Indices. *ACM Trans. Database Syst.* **12**(2), 218–246 (Jun 1987)
23. Vicknair, C., et al.: A comparison of a graph database and a relational database: A data provenance perspective. *Proceedings of the 48th Annual Southeast Regional Conference* (2010)