



PDF Download
3722212.3724434.pdf
09 February 2026
Total Citations: 0
Total Downloads: 886

Latest updates: <https://dl.acm.org/doi/10.1145/3722212.3724434>

RESEARCH-ARTICLE

Dynamic Pruning for Recursive Joins

NORIFUMI NISHIKAWA, Hitachi, Ltd., Tokyo, Japan

AKIRA SHIMIZU, Hitachi, Ltd., Tokyo, Japan

AKIRA ITO, Hitachi, Ltd., Tokyo, Japan

SHINJI FUJIWARA, Hitachi, Ltd., Tokyo, Japan

YUTO HAYAMIZU, The University of Tokyo, Tokyo, Japan

M. KITSUREGAWA, The University of Tokyo, Tokyo, Japan

[View all](#)

Open Access Support provided by:

The University of Tokyo

Hitachi, Ltd.

Published: 22 June 2025

[Citation in BibTeX format](#)

SIGMOD/PODS '25: International
Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

Dynamic Pruning for Recursive Joins

Norifumi Nishikawa
Hitachi, Ltd.
Kokubunji, Tokyo, Japan
norifumi.nishikawa.mn@hitachi.com

Akira Shimizu
Akira Ito
Shinji Fujiwara
Hitachi, Ltd.
Yokohama, Kanagawa, Japan

Yuto Hayamizu
Masaru Kitsuregawa
Kazuo Goda
The University of Tokyo
Meguro-ku, Tokyo, Japan

Abstract

This paper studies recursive join queries, which are witnessed in recent industrial applications such as event sequence analysis and traceability analysis. The recursive join query has the possibility to invoke deeply nested joins. Known pruning techniques generating pruning filters at compile time may not be able to exploit pruning opportunities, in particular for deep joins, at full. This paper explores *dynamic pruning* for recursive joins. Dynamic pruning generates pruning filters during query execution; a pruning filter is made based on the result of each recursive step, and it helps to more significantly reduce the amount of relation records to be accessed at the next recursive step, hence improving the query performance. This paper presents the intensive and extensive experiments that we have conducted with a commercial prototype database engine to clarify the performance benefit of dynamic pruning with four business benchmark scenarios. Our observation demonstrates that dynamic pruning significantly speeds up recursive join queries and non-recursive join queries having deeply nested joins; it works up to 135.6 times faster than the baseline case and 3.99 times faster even than the state-of-the-art pruning approach. In addition, our concise filter design only yielded negligibly small processing overheads for ineffective queries, even though the filter was intensively manipulated at runtime.

CCS Concepts

• Information systems → Database query processing; Online analytical processing engines; Physical data models.

Keywords

Dynamic pruning, Recursive query, Traceability

ACM Reference Format:

Norifumi Nishikawa, Akira Shimizu, Akira Ito, Shinji Fujiwara, Yuto Hayamizu, Masaru Kitsuregawa, and Kazuo Goda. 2025. Dynamic Pruning for Recursive Joins. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25)*, June 22–27, 2025, Berlin, Germany. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3722212.3724434>

1 Introduction

Recursion is a fundamental idea widely seen in data structures and search algorithms [22]. How to incorporate recursion into database queries was actively explored in the deductive database study in the

1980s [3, 9, 11, 35, 77, 78, 82, 96]. To date, many commercial database engines hold the processing capability of recursive queries [76]. The query optimization on recursive queries was not necessarily actively studied in the academic community [69], but it is becoming a lively topic due to the recent demand for complicated query processing [6, 38, 39, 42, 98, 100, 102, 111].

We have explored the technical opportunity of more intensively pruning the processing work necessary for recursive joins by exploiting execution information available at runtime. This attempt has been motivated by recent emerging workloads, such as event sequence analysis [37, 44] and traceability analysis [14, 50, 53], since they are naturally described by recursive queries.

This paper studies *dynamic pruning* that generates and propagates pruning filters during query execution. Dynamic pruning exploits the runtime available information for building the filter. Specifically, a separate filter is allowed to be made based on the result of each recursive step, and it helps to more significantly reduce the amount of relation records to be accessed at the next recursive step. Hence, dynamic pruning potentially improves the query performance, particularly of recursive join queries having deeply nested joins [10, 20, 68, 69, 75, 79].

This paper presents our experiments to clarify the performance benefit of dynamic pruning. We have prototyped dynamic pruning on HADB, Hitachi's commercial database engine [40], and conducted comparative studies with standard decision-making benchmarks (TPC-H [93] and TPC-DS [92]), a manufacturing business database benchmark (4mbench [29]) and citation network analysis (cit-analysis). TPC-H and TPC-DS have the benefit of being utilized in many database performance studies, but their original queries have no recursive joins. Thus, we have appended four recursive queries of event sequence analysis that work on the TPC-H schema and dataset. 4mbench contains three recursive queries that simulate traceability analysis jobs on the manufacturing event history. cit-analysis is composed of five recursive queries [83]. The experiments have confirmed that dynamic pruning substantially speeds up recursive join queries and even some non-recursive join queries, specifically performing up to 135.6 times faster than the baseline case and 3.99 times faster even than the state-of-the-art pruning approach. Interestingly, it yielded only negligibly small performance penalties for queries for which the pruning is not effective. A vision for applying runtime pruning techniques to complicated queries was addressed in recent work [13, 21, 51, 55, 65]. The contributions of this paper can be summarized as follows.

- This paper explores the unique opportunities brought by *dynamic pruning* for recursive joins that have been witnessed in recent industrial applications.



This work is licensed under a Creative Commons Attribution 4.0 International License. *SIGMOD-Companion '25, Berlin, Germany*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1564-8/2025/06
<https://doi.org/10.1145/3722212.3724434>

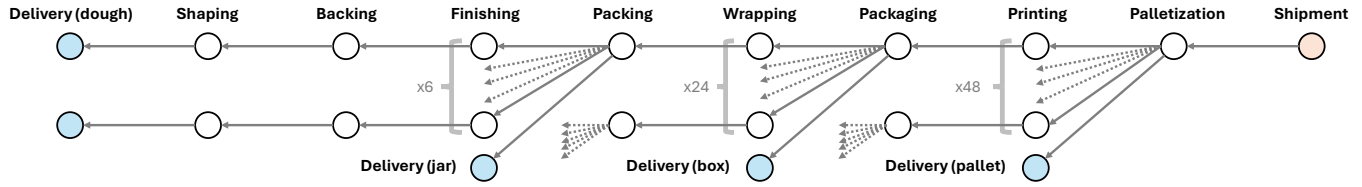


Figure 1: A case of backward traceability in the food processing business [46, 59]. Each arrow indicates a transition step in the traceability analysis. Starting from a processing event (indicated by the right-most pink circle) that produces a final product, the query recursively traces back each of its associated events until it reaches all the associated primordial events (i.e., blue circles).

- This paper formalizes a specific algorithm for dynamic pruning to enable the transitive propagation of the pruning filter among different operators to reduce the IO amount, and its implementation techniques that we have employed to prototype it on a commercial database engine.
- This paper presents the intensive and extensive experiments that we have conducted with the commercial prototype to clarify the performance benefit of dynamic pruning with four business benchmark scenarios in comparison with other pruning techniques.

The remainder of this paper is organized as follows. Section 2 presents the motivation for this study and reviews the related work. Section 3 presents a formal definition of recursive join queries and a brief survey of its major algorithms. Section 4 explores dynamic pruning in contrast with existing pruning techniques. Section 5 presents our experimental study to clarify the performance benefit of dynamic pruning. Finally, Section 6 concludes the paper.

2 Motivation and related work

2.1 Motivation

Let us motivate the study of the query optimization of recursive joins by giving two specific applications.

One major application is *event sequence analysis* [37, 44]. Suppose a retail history database that stores customer orders. Business owners often try to figure out ordering patterns that are specific to customers of high loyalty because they are helpful in designing business improvement plans. Let us suppose a business query to enumerate customers who ordered Z after ordering X once and then ordering Y more than three times but placed no other orders between the order of X and the order of Z; expected customers are supposed to have ordering patterns like state transitions $X \rightarrow Y \rightarrow Y \rightarrow \dots \rightarrow Y \rightarrow Z$ in chronological order. One possible query definition describes the iterative part by using the recursion; specifically, it verifies the transition $Y \rightarrow Y$ recursively after the transitions $X \rightarrow Y$ and then verifies the transition $Y \rightarrow Z$. Similar queries are seen in other business scenarios on clinical episode analysis [36, 60, 97] and online game activity analysis [17, 32].

Another major application is *traceability analysis* [14, 50, 53]. Suppose a manufacturing business database that stores every processing event in a manufacturing firm. A capability of exhaustive production inspection or bill-of-materials inspection potentially reduces the delivery of defective products by thoroughly verifying the manufacturing history before shipping every product. Its query often involves a recursive step of tracing back the manufacturing

history from a final product (or intermediate product) to each of its previous products. Figure 1 illustrates the processing of such a backward traceability query. Another expected capability is focused defect influence investigation; in case that a defective status is detected in a specific machine in a production pipeline, this capability walks through the manufacturing history to identify all the intermediate and final products that are possibly affected by the defect. The query involves a recursive step of tracing forward the production dependency history. Traceability workloads are seen in other applications such as social network analysis [5, 7] and software security management [24, 45].

A practical software solution for enabling this type of analysis has been to employ a specialized technology like graph database engines that have the capability of obtaining transitive closures efficiently [23, 70, 88, 111]. Another possible solution is to allow relational database engines to run this type of analytical query efficiently in order to consolidate different analytical workloads into one place. The recursive query is by no means new for relational database engines. Modern commercial database engines support the recursion, being able to answer recursive queries [76]. Yet, the performance optimization of the recursive query has not been actively explored [38, 69].

Dynamic pruning is our solution for relational database engines to improve the execution efficiency of recursive join queries. The basic idea of dynamic pruning is to generate a pruning filter based on the result of each recursive step and to leverage the filter to reduce the amount of relation records to be accessed at the next recursive step during query execution. Being dynamically generated in the recursion loop, the filter remains effective even at deep recursive steps. Our experiments with a commercial prototype, later presented in Section 5, have demonstrated that dynamic pruning performs better than the state-of-the-art pruning technique for recursive join queries.

2.2 Related work

The idea of passing *sideways* information (i.e., data other than output tuples) from one portion to another portion during query execution was attempted in early studies on distributed database and deductive database in order to improve the efficiency of query processing. Bernstein et al. presented the semi-join operator to allow database nodes in the distributed environment to produce and exchange projected attributes before performing a join operation [12]; the exchanged attributes helped each node to reduce the amount of tuples to be shipped to another node via the network for the join [49, 57, 107]. Mackert et al. presented another idea of

using Bloom filters instead of projected attributes [58]. To date, the Bloom filter and its variations, hash-based filters, have been actively explored [18, 41, 62, 73, 90]. Typically, these filters are exploited in join and join-like operations. The filter is generated from the outer block (e.g., a build phase in hash join or a main subexpression invoking a subquery), and it is then propagated to the inner block (e.g., a probe phase or an invoked subquery); the inner block leverages the filter to prune its output tuples to be shipped back to the outer block, thus reducing the computation cost of the join. These techniques have been employed in recent commercial database engines [1, 26, 84]. Yet another filtering technique was magic sets rewriting, presented by Bancilhon et al. [9] for declarative logic programming. Similarly, it identifies the possible value bound from the outer block (originally, a DataLog rule) and propagates it to the inner block to reduce the costly work of the main query [11]. This idea was later extended to SQL queries [63, 81].

These runtime pruning techniques work elegantly to reduce the computation of the join and join-like operations. In contrast, dynamic pruning presented in this paper reduces the IO amount to be issued for the inner block by merging the pruning filter generated from the outer block with the summary information on the storage layout. The idea of combining the pruning filter and the summary information to reduce IOs was recently presented by Kandula et al. [71]. Their design philosophy was to intensively exploit the predicate information and the summary information at compile time (i.e., instead of at runtime), while relieving the runtime overhead by simplifying the execution mechanism. Our study has demonstrated that the dynamic pruning approach performs much better for recursive join queries with negligibly small overheads induced by runtime filter generation and evaluation.

An early paper authored by Stocker et al. presented a potential benefit of applying the semi-join operator to reduce the IO amount [85]. Nica et al. presented an idea of extending the runtime partition pruning technique (skipping unnecessary partitions) to partition-wise joins and reported the potential pruning effects [65]. Several papers on commercial/open-source implementation addressed a similar idea as a design option [13, 21, 51, 55]. To our knowledge, the unique contributions of our paper are that we have formalized an algorithm for dynamic pruning to enable the transitive propagation of the pruning filter among different operators to reduce the IO amount, and we have experimentally demonstrated its pure performance benefits for recursive joins and deeply nested joins on a commercial database engine.

Ives et al. introduced the adaptive approach to pass the sideways information between separate joins mutually correlated through predicates for push-style query processing [41]. Neumann et al. extended this approach to concurrently running joins by passing the on-the-fly information on RDF graph processing [64]. Our approach uses only the completed result of every join step, offering significant performance benefits for recursive queries.

Several recent query optimization studies have again shed light on the DataLog engine [6, 39, 42, 98, 111]. Wang et al. proposed a rewriting rule, called the FHG rule, for optimizing recursive DataLog programs with query synthesis [100]. Wu et al. developed a parallel evaluation framework for recursive DataLog programs tuned for the in-memory multicore environment [102]. Herlihy et

Algorithm 1 Seminaïve algorithm.

```

1:  $R_1 \leftarrow \sigma_{p_S}(S) \bowtie_{S,y=T.x} \sigma_{p_T}(T)$  ▷ Anchor step
2:  $R \leftarrow R_1$ 
3: for  $d \leftarrow 1; |R_d| > 0; d \leftarrow d + 1$  do ▷ Recursive steps
4:    $R_{d+1} \leftarrow R_d \bowtie_{R_d.j=T.i} \sigma_{p'_T}(T)$ 
5:    $R \leftarrow R \cup R_{d+1}$ 
6: return  $R$ 

```

al. proposed an adaptive mechanism for enabling runtime query re-optimization and code regeneration [38]. Our study has focused on the runtime pruning mechanism to reduce the IO amount necessary for relational recursive queries.

3 Recursive join

This section formally defines a recursive join query and reviews its algorithms that work on relational database engines.

3.1 Problem definition

A recursive query is a query containing recursion in its definition. The study on recursive queries can be traced back to the deductive database technology that was actively studied to combine logical programming and relational database in the 1980s [3, 9, 11, 35, 77, 78, 82, 96]. A variety of forms are theoretically possible in recursive queries. This paper studies a *recursive join query* [10, 20, 68, 69, 75, 79] of the following form to fit in the emerging business demand discussed in the previous section.

Definition 3.1. (Recursive join query) A recursive join is defined as an operation to obtain R satisfying:

$$\begin{aligned}
 R_1 &= \sigma_{p_S}(S) \bowtie_{S,y=T.x} \sigma_{p_T}(T) \\
 R_{d+1} &= R_d \bowtie_{R_d.j=T.i} \sigma_{p'_T}(T) \quad (d \geq 1) \\
 R &= \bigcup R_d
 \end{aligned}$$

from input relations S and T , where $p_S(s)$, $p_T(t)$ and $p'_T(t)$ are the predicates of a tuple s in S and a tuple t in T , respectively. A *recursive join query* refers to a query containing the recursive join.

So far, major work on recursive join queries used a simplified definition that considers a single relation T for recursion and joins T with itself recursively [68, 69, 79]. We have, however, chosen the above definition that extends the simplified definition since a distinction between a relation S for the anchor step and a relation T for the recursive steps is essential for business queries that we have observed in industry applications. Obviously, assuming that S is a unit relation that satisfies $T = S \bowtie T$ and that $p_S(s)$, $p_T(t)$, $p'_T(t)$ always return true, our definition is equivalent to the simplified definition.

3.2 Algorithms to evaluate recursive join queries

Known algorithms to evaluate recursive join queries can be grouped into two approaches, *seminaïve* and *direct*. The seminaïve algorithm is a straightforward bottom-up solution, being composed of an *anchor step* for obtaining R_1 and one or more *recursive steps* for obtaining R_{d+1} ($d \geq 1$) [10]. Algorithm 1 presents a pseudo code,

Algorithm 2 Static pruning.

```

1:  $y_0, y_1 \leftarrow \text{FINDPRUNINGRANGE}(S, y)$ 
2:  $T'_1 \leftarrow \text{PRUNESegMENTS}(T, x, y_0, y_1)$ 
3: for  $d = 1; d \leq N - 1; d++$  do
4:    $j_0, j_1 \leftarrow \text{FINDPRUNINGRANGE}(T'_d, j)$ 
5:    $T'_{d+1} \leftarrow \text{PRUNESegMENTS}(T, i, j_0, j_1)$ 
6:    $R_1 \leftarrow \sigma_{p_S}(S) \bowtie_{S.y=T'_1.x} \sigma_{p_T}(T'_1)$  ▷ Anchor step
7:    $R \leftarrow R_1$ 
8:   for  $d \leftarrow 1; |R_d| > 0; d \leftarrow d + 1$  do ▷ Recursive steps
9:      $R_{d+1} \leftarrow R_d \bowtie_{R_d.j=T'_{d+1}.i} \sigma_{p'_T}(T'_{d+1})$ 
10:     $R \leftarrow R \cup R_{d+1}$ 
11: return  $R$ 

// Find a pruning range for attribute  $a$  of relation  $R$ .
12: function  $\text{FINDPRUNINGRANGE}(R, a)$ 
13:   if min/max statistics of  $a$  in  $R$  is not available then
14:     return  $-\infty, \infty$ 
15:    $k_{\min}, k_{\max} \leftarrow \infty, -\infty$ 
16:   for all segment  $s$  in  $R$  do
17:      $a_0, a_1 \leftarrow$  attribute  $a$  min/max from statistics of  $s$ 
18:      $k_{\min} \leftarrow \min(k_{\min}, a_0), k_{\max} \leftarrow \max(k_{\max}, a_1)$ 
19:   return  $k_{\min}, k_{\max}$ 

// Prune segments of  $R$  with a range  $[k_{\min}, k_{\max}]$  of attribute  $a$ .
20: function  $\text{PRUNESegMENTS}(R, a, k_{\min}, k_{\max})$ 
21:   if min/max statistics of  $a$  in  $R$  is not available then
22:     return  $R$ 
23:    $R' \leftarrow \emptyset$ 
24:   for all segment  $s$  in  $R$  do
25:      $a_0, a_1 \leftarrow$  attribute  $a$  min/max from statistics of  $s$ 
26:     if  $\neg(k_{\max} < a_0 \vee a_1 < k_{\min})$  then
27:        $R' \leftarrow R' \cup \{s\}$ 
28:   return  $R'$ 

```

Algorithm 3 Dynamic pruning.

```

1:  $\overline{S'}, y_0, y_1 \leftarrow \text{SCANANDFINDRANGE}(S, y, p_S)$ 
   ▷  $\overline{S'}$  may have scan processing states (e.g. hash build table).
2:  $T'_1 \leftarrow \text{PRUNESegMENTS}(T, x, y_0, y_1)$ 
3:  $R_1 \leftarrow \overline{S'} \bowtie_{S'.y=T'_1.x} \sigma_{p_T}(T'_1)$  ▷ Anchor step
4:  $R \leftarrow R_1$ 
5: for  $d \leftarrow 1; |R_d| > 0; d \leftarrow d + 1$  do ▷ Recursive steps
6:    $\overline{R'_d}, j_0, j_1 \leftarrow \text{SCANANDFINDRANGE}(R_d, j, p(r) \equiv \top)$ 
7:    $T'_{d+1} \leftarrow \text{PRUNESegMENTS}(T, i, j_0, j_1)$ 
8:    $R_{d+1} \leftarrow \overline{R'_d} \bowtie_{R'_d.j=T'_{d+1}.i} \sigma_{p'_T}(T'_{d+1})$ 
9:    $R \leftarrow R \cup R_{d+1}$ 
10: return  $R$ 

// Scan  $\sigma_{p(r)}(R)$  while finding a pruning range for attribute  $a$ 
11: function  $\text{SCANANDFINDRANGE}(R, a, p(r))$ 
12:    $k_{\min}, k_{\max} \leftarrow \infty, -\infty$ 
13:    $\overline{R'} \leftarrow \emptyset$  with scan processing states (e.g. hash build table)
14:   for all segment  $s$  in relation  $R$  do
15:     for all record  $r$  in segment  $s$  do
16:       if  $p(r)$  then
17:          $k_{\min}, k_{\max} \leftarrow \min(k_{\min}, r.a), \max(k_{\max}, r.a)$ 
18:         Add  $r$  to  $\overline{R'}$ 
19:   return  $\overline{R'}, k_{\min}, k_{\max}$ 

```

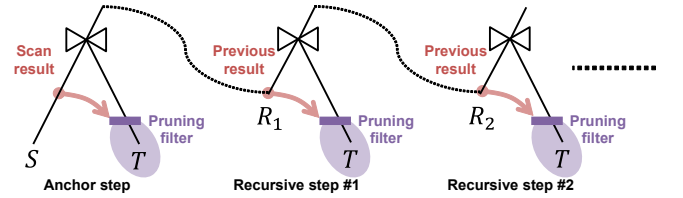


Figure 2: Dynamic pruning for recursive joins.

where intermediate relations, R_1 and R_d , are supposed to be stored as temporary relations in the database. For simplicity, the algorithm assumes the equi-join, but it can be directly applied to other joins. For example, Section 4.4 will present a case with inequality expressions. Although the presented code allows the recursion to continue until it converges for simplicity, the code may be adjusted to limit the recursion by a certain boundary. Another solution is the direct algorithm that uses the spatial structure of a transitive closure problem, being able to complete in a fixed number of steps but potentially yielding redundant processing [3, 43, 80, 101].

The remainder of this paper focuses on the seminaïve algorithm for discussing the pruning technology because it often performs better for business workloads that this paper assumes and it has been studied in other papers [68, 70].

4 Dynamic pruning for recursive joins

This section reviews existing pruning techniques that potentially work for general selection and join queries. Then, we present our solution, *dynamic pruning*, to improve the execution performance of recursive join queries.

4.1 Summary information and trivial pruning

Modern database engines manage the summary or statistical information of stored datasets and their organization [15, 65, 86, 91]. Typically, the summary information contains ranges (i.e., minimum and maximum values) for possible key attributes of relations in each segment in the database [31, 47, 52, 61, 89, 112, 113]. This paper uses a term, *segment*, to refer to a database part for which the range information is managed. The definition of segment varies with implementation; it may be a physical unit (e.g., a fixed number of consecutive pages) or a logical unit (e.g., a database partition storing records satisfying a specific condition). An example of the summary information will be presented in Section 4.4 and Figure 3.

It is known that, coupled with predicate pushdown, the summary information potentially allows a scan operator in a query to perform more efficiently [56, 65, 87, 106]. Normally, the scan operator visits every database segment, in which it scans or searches records in a target relation and transfers only records satisfying the predicate condition (described in a query) to its next operator. The summary information enables the scan operator to make a decision about whether to visit every segment by evaluating the compatibility

between the range information of the segment and the predicate condition. The scan operator is allowed to skip the scanning job for segments, in each of which it is confirmed no possible records satisfying the predicate condition are stored. This paper calls this approach *trivial pruning* since the filtering condition is trivial to the optimizer. Recent parallel/cloud platforms have eagerly employed this pruning technique [4, 8, 19, 48, 103].

The join operation is binary. The trivial pruning technique is independently applicable to a scan operator connected to each side of the join operation [16, 27, 33, 34, 67, 72], and it is the case even though the join is recursive. When it comes to the seminaïve algorithm (Algorithm 1), the scan operators may be able to exploit the predicates $p_S(s)$ for S , $p_T(t)$, $p'_T(t)$ for T with the summary information to skip incompatible segments, thus potentially reducing the scan work and improving the query performance.

4.2 Static pruning

It is known that the pruning opportunity for the join operation can be extended in consideration of the dependency induced by the join predicate. One major approach is to generate a pruning filter from the summary information on a left-handed (i.e. firstly scanned) relation and the query predicate at compile time, and then to apply the pruning filter to the scan job of a right-handed (i.e. secondly scanned) relation during query execution [71].

Suppose that two relations R and S are stored in the database, and the summary information reserves range values on $R.r$ and $R.x$ for R and those on $S.s$ for S . Let us think about a join operation, $\sigma_{R.x(R)} \bowtie_{R.r=S.s} S$, where the scan job on S follows the scan job on R . Merging the selection condition given at $\sigma_{R.x}$ and the summary information (i.e., range values on $R.r$ and $R.x$) can yield a pruning filter describing the ranges in $R.r$ in which records possibly exist in R . Further merging the join predicate $R.r = S.s$ and the previous filter can yield another pruning filter describing the ranges of $S.s$ in which records possibly exist. This calculation is feasible at compile time [2]. The resulting pruning filter may be leveraged in the right-handed scan of the join operation, being likely to offer a better pruning effect for the join operation than trivial pruning. This paper calls this pruning approach *static pruning* since a pruning filter is statically determined at compile time.

Static pruning is also directly applicable to recursive joins. Algorithm 2 presents a revised code of the seminaïve algorithm with static pruning. Lines 1 to 5 are a preprocessing part to generate pruning filters ($[y_0, y_1]$ for the anchor step and $[j_0, j_1]$ for each iteration of the recursive step) and apply these filters to obtain pruned relations $T'_d (d = 1, \dots)$ at compile time and Lines 6 to 11 are a runtime part to process scan and join operators with pruned relations T'_d .

4.3 Dynamic pruning

Let us further extend the pruning opportunity for the join operation by generating the pruning filter *during query execution* rather than at compile time. The basic idea is to generate a pruning filter directly from the scan job of a left-handed (i.e., firstly scanned) relation and the query predicate during query execution and apply the filter to the scan job of a right-handed (i.e. secondly scanned) relation.

We have named this idea *dynamic pruning* since a pruning filter is dynamically determined while the query is being executed.

Again, suppose a join operation, $\sigma_{R.x(R)} \bowtie_{R.r=S.s} S$. Dynamic pruning summarizes the scan result of $\sigma_{R.x} R$ during query execution to yield a pruning filter along the $R.r$ key space, and then it converts the pruning filter along the $S.s$ key space. In contrast to static information that leverages the pre-determined summary information, dynamic pruning exploits the runtime information, which allows the query processing to obtain a more precise pruning filter and thus achieves a better pruning effect, particularly for deeply nested joins.

Algorithm 3 presents a revised code of the seminaïve algorithm where dynamic pruning is incorporated. A separate pruning filter is generated at runtime ($[y_0, y_1]$ at Line 1 and $[j_0, j_1]$ at Line 6) and applied for the scan job (Lines 2 to 3 and Lines 7 to 8), where the filter is coupled with the summary information, identifying segments that can be skipped for the scan job. Figure 2 illustrates how the pruning filter is generated and propagated.

4.4 Case study

So far, we have presented two pruning approaches, static pruning and dynamic pruning, which help to improve the execution performance of recursive join queries. Here, we present a case study to demonstrate that dynamic pruning offers better pruning effects than static pruning.

Figure 3 illustrates an example case¹. Suppose that the database has two relations, CUSTOMER and ORDERS, and the ORDERS records are partitioned among four separate segments for each month of ODATE; this record distribution is managed in the summary information.

Let us think about a recursive join query (tracing forward the order history of a specific customer) as follows:

$$\begin{aligned} R_1 &= \sigma_{p_1(C, O)}(C \bowtie_{C.CID=O.CID} O) \\ \text{where } p_1(C, O) &\equiv C.CNAME = \text{'Jimmy Kudo'} \\ &\quad \wedge O.ODATE \in [1994-01-01, 1994-01-15] \\ R_{d+1} &= R_d \bowtie_{p_2(O, R_d)} O \\ \text{where } p_2(O, R) &\equiv R.CID = O.CID \\ &\quad \wedge O.ODATE \in [R.ODATE + 1_{\text{day}}, R.ODATE + 1_{\text{month}}] \\ R &= \bigcup R_d \end{aligned}$$

where CUSTOMER and ORDERS are abbreviated to C and O , respectively.

Consider how static pruning works for this query. Note that static pruning generates a pruning filter by using the query predicate and the summary information at compile time. (1) Looking into the anchor step for obtaining R_1 , the selection and join predicate derives a pruning filter indicating that the compatible ORDERS records exist only in the ODATE space of $[1994-01-01, 1994-01-15]$. Being coupled with the summary information, the query optimizer is informed that the scan operator on ORDERS has to visit only segment #1 and it can skip segments #2 to #4. (2) When it comes to the first recursive step for obtaining R_2 , by applying the join predicate

¹Obviously, the example case follows a simplified schema of TPC-H [93] because this is familiar to the database technology community.

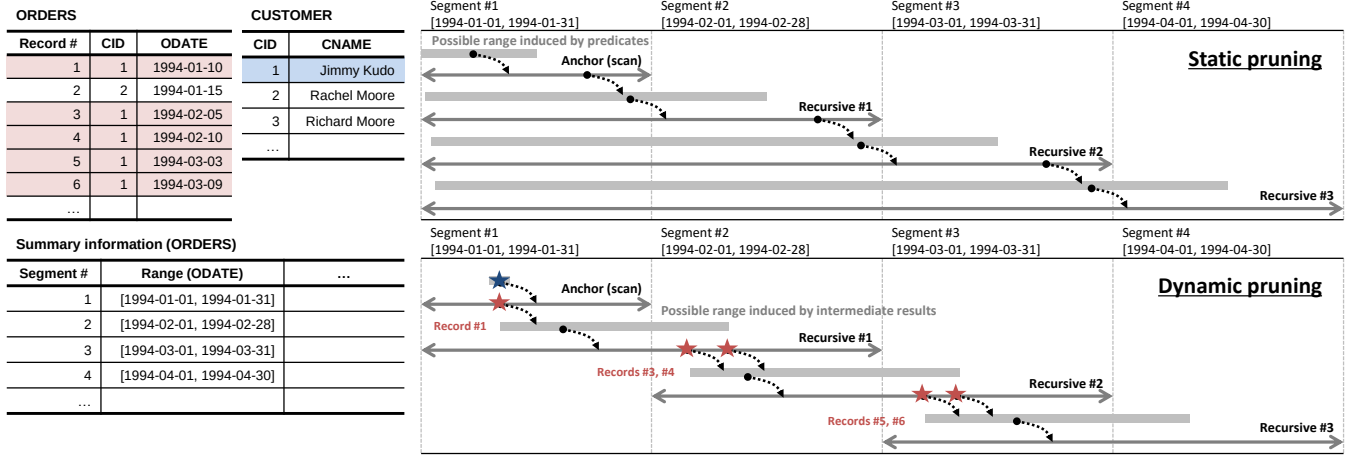


Figure 3: A comparative case study of static pruning and dynamic pruning for recursive joins

to the pruning filter derived in the anchor step, the query compiler can obtain a next pruning filter indicating that the compatible ORDERS records exist only in [1994-01-02, 1994-02-15]. This range corresponds to segments #1 and #2; the query compiler invokes the scan job on segments #1 and #2. (3) The remaining steps work similarly. In the second recursive step for R_3 , further application of the join predicate yields a pruning filter of [1994-01-03, 1994-03-15]. The scan operator has to visit segments #1 to #3 since they are corresponding to this range. In the third recursive step, the pruning filter becomes [1994-01-04, 1994-04-15]; thus the scan operation is invoked on the corresponding segments #1 to #4.

This example implies that static pruning propagates a pruning filter beyond every join operation and thus offers effective pruning performance for recursive joins. However, the pruning effect degrades at deep recursive steps. Static pruning only generates pruning filters by using the summary information and the given query predicates at compile time. Clues available at compile time are by no means sufficient to fully exploit the pruning opportunity at deeply nested joins.

In contrast, dynamic pruning generates a pruning filter during query execution, exploiting the intermediate result to sophisticate the pruning. (1) Again, looking into the anchor step, the scan result of CUSTOMERS derives a pruning filter indicating that the compatible ORDERS records exist only at the ODATE value of 1994-01-10. Being coupled with the summary information, the query optimizer is informed that it needs to invoke the scan job only on segment #1 and it can skip segments #2 to #4. Hence, the anchor step obtains R_1 having Record #1 (1994-01-10). (2) In the first recursive step for obtaining R_2 , the query executor applies the join predicate to the range of R_1 to offer a next pruning filter indicating that the compatible ORDERS records exist only in [1994-01-11, 1994-02-10]. This range corresponds to segments #1 and #2; the query compiler invokes the scan on segments #1 and #2. As a result, this step obtains R_2 having Records #3 and #4 ([1994-02-05, 1994-02-10]). (3) The remaining steps work similarly. In the second recursive step for R_3 , further application of the join predicate to R_2 yields a pruning filter of [1994-02-06, 1994-03-10]. The scan operator has to visit segments

#2 to #3 since they correspond to this range. This step finally obtains R_3 having Records #5 and #6 [1994-03-03, 1994-03-09]. In the third recursive step, the pruning filter becomes [1994-03-04, 1994-04-09]; thus the scan operation is invoked only on the corresponding segments #3 to #4.

This study indicates that dynamic pruning offers better pruning effects than static pruning, particularly performing successfully at deep recursive steps. This benefit comes from its unique capability of generating pruning filters by utilizing the runtime information such as intermediate results. Specifically, dynamic pruning is able to generate a pruning filter from the result of its previous step.

For simplicity, this example assumes that different segments are responsible for disjoint ranges. The pruning mechanism works if different segments overlap in range.

4.5 Implementation

We have prototyped dynamic pruning in Hitachi's commercial database engine, HADB [40]. This section summarizes our implementation strategy.

HADB manages the summary information for each relation at two levels. Specifically, it embeds *range index* [61, 89, 113], a concise data structure storing minimum and maximum values for concerned key attributes, in every *chunk* (i.e., a table partition of records satisfying a specific range condition) and every *extent* (i.e., a fixed number of consecutive pages) for each relation. A chunk is a larger allocation unit, being composed of multiple extents. At every query block (i.e., a series of pipelined operators), HADB generates a pruning filter by checking each outgoing record and keeping minimum and maximum values for each attribute appearing in the predicate to be evaluated at its next query block. At its next query block, HADB combines the pruning filter and the summary information to identify chunks and extents to be actually scanned.

In this way, we have employed a very concise filter design (only managing a single range for each attribute in the summary information and the pruning filter), but intensive filter generation and evaluation (generating a filter by checking each outgoing record and evaluating it for each chunk and extent). As shown in Section 5,

Table 1: Recursive join queries on TPC-H and 4mbench.

(a) TPC-H	
RQ1	Enumerates the high-loyalty customers having placed continuous orders.
RQ2	Enumerates the order histories of customers having placed orders in a specific purchasing pattern.
RQ3	Enumerates the customers having bought products of a specific brand repeatedly.
RQ4	Makes a list of popular products that have been continuously sold within a specific duration.
(b) 4mbench	
Q3	Calculates the distribution of production lead time.
Q4	Verifies the normality of all ancestor processing events associated with a specific final product.
Q5	Enumerates all the final products possibly affected by specific defective materials.

this strategy works well; it offers significant speedups for many recursive join queries and deeply-nested join queries, but it only yields negligibly small overheads for ineffective queries.

5 Experiments

This section presents our intensive and extensive experiments to clarify the performance benefit of dynamic pruning for analytical queries on a commercial database management system on a modern enterprise machine. Our observation indicates that dynamic pruning speeds up recursive join queries and even some non-recursive join queries more significantly than other pruning techniques.

5.1 Experimental setup

5.1.1 Test cases and business workloads. For a comparative study, we prototyped the five pruning test cases as follows in HADB 05-07/A, Hitachi’s commercial database management system tuned for analytical workloads [40]. **Baseline** corresponds the basic algorithm, Algorithm 1, with trivial pruning. Specifically, this case does not generate or propagate any pruning filters in a recursive join process. It only utilizes a trivial predicate to skip segments. **Static-A** generates a pruning filter for an anchor step in a recursive join process at compile time and then propagates it during execution. This case partially implements Algorithm 2. **Static-AR** (static pruning) generates pruning filters for anchor and recursive steps in a recursive join process at compile time and then propagates them during execution. This case is equivalent to Algorithm 2, the state-of-the-art pruning approach [71]. By nature, the necessary recursive depth (i.e., N at Line 3) is unknown before query execution. For comparison fairness, we conducted preliminary tests in advance for every test query to figure out the recursive depth and utilized it for performance tests so that we could observe the tuned performance. **Dynamic-A** generates and propagates a pruning filter for an anchor step in a recursive join process during execution. This case partially implements Algorithm 3. **Dynamic-AR** (dynamic pruning) generates and propagates pruning filters for anchor and

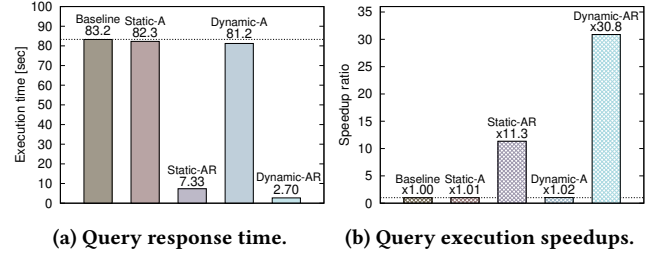


Figure 4: Execution performance comparison among different pruning techniques for a test recursive join query (RQ1) on the TPC-H dataset (scale factor: 1,000). *Dynamic pruning of recursive joins (Dynamic-AR) speeds up the query execution by 30.8 times than the baseline case (Baseline) and 2.72 times even than the state-of-the-art technique (static pruning, Static-AR).*

recursive steps in a recursive join process during execution. This case is equivalent to Algorithm 3. Obviously, **Dynamic-AR** denotes the full-fledged implementation of dynamic pruning. Note that we implemented these pruning techniques in a generalized fashion so that they can work for non-recursive joins. This section highlights the performance difference among **Dynamic-AR** and the others.

We prepared four different analytical database workloads and utilized them for performance evaluation. The first and second workloads that we chose were TPC-H [93] and TPC-DS [92], standard analytics database benchmarks. TPC-H has the great benefit of having been used in many database studies and industrial tests. However, its workload can be traced back to the industry experience in the 1990s; their twenty-two standard queries do not consider or contain recursive join operators that this paper focuses on. TPC-DS is a newer benchmark with higher complexity, but it still contains no recursive joins. Hence, we designed four additional test queries (RQ1 to RQ4) containing recursive join operators that directly work on the standard TPC-H schema and dataset². Table 1(a) summarizes their business questions. The third workload was 4mbench, a database benchmark simulating an emerging business analytics workload in the manufacturing industry [29, 94]. 4mbench defines six test queries in total; three (Q3, Q4, and Q5) of them involve recursive join operators to perform traceability investigations throughout the history of manufacturing events. Table 1(b) summarizes their business questions. The fourth workload was citation network analysis (cit-analysis). Specifically, we converted five recursive DataLog-based queries (i.e., TC (transitive closure), SG (same generation), Reachability (REACH), SSSP (single-source shortest path) and CC (connected components)) [83] to SQL codes and tested them on the cit-Patents dataset downloaded from Stanford Network Analysis Project [54]. SQL codes and scripts utilized in the experiments are available on the public repository [95].

For performance measurement, we first generated a dataset for every test configuration and loaded it into the database. By default, the dataset was generated for TPC-H and TPC-DS with a scale factor of 1,000 (approx. 1,081 GB and 917 GB) and for 4mbench with

²We generated these four business queries by extending event sequence queries that we designed on the TPC-H schema and dataset in our past work [37].

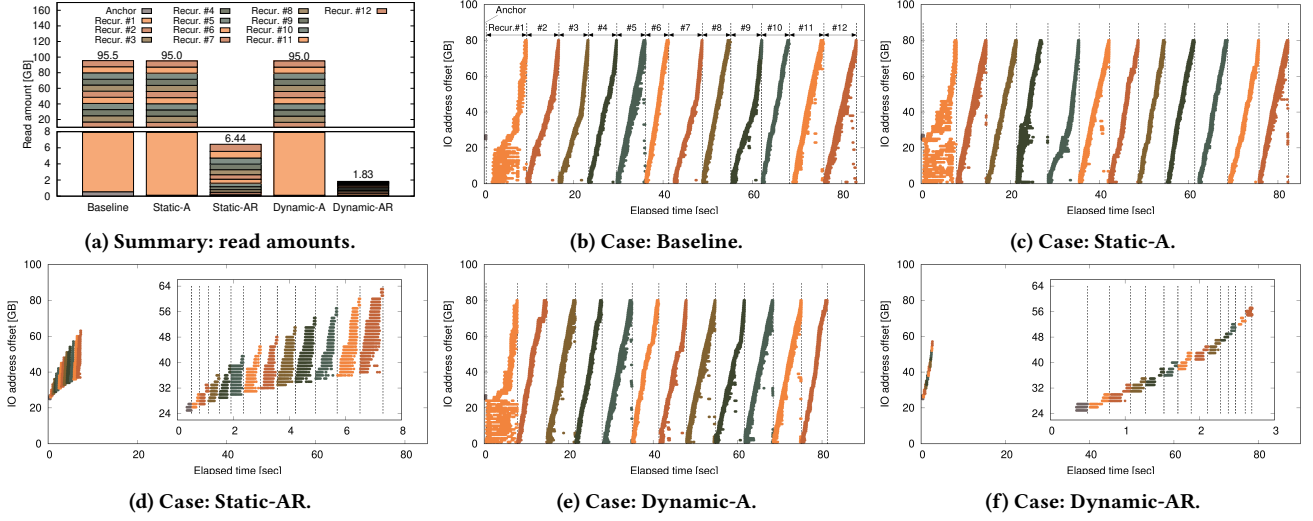


Figure 5: (a) Read amount comparison among different pruning techniques for a test recursive join query (RQ1) on the TPC-H dataset (scale factor: 1,000), and (b-f) their traces of data read commands on ORDERS. *Dynamic pruning of recursive joins (Dynamic-AR) reduces data reads necessary for the query execution of RQ1 by 52.2 times than the baseline case (Baseline) and 3.52 times even than the state-of-the-art technique (static pruning, Static-AR).*

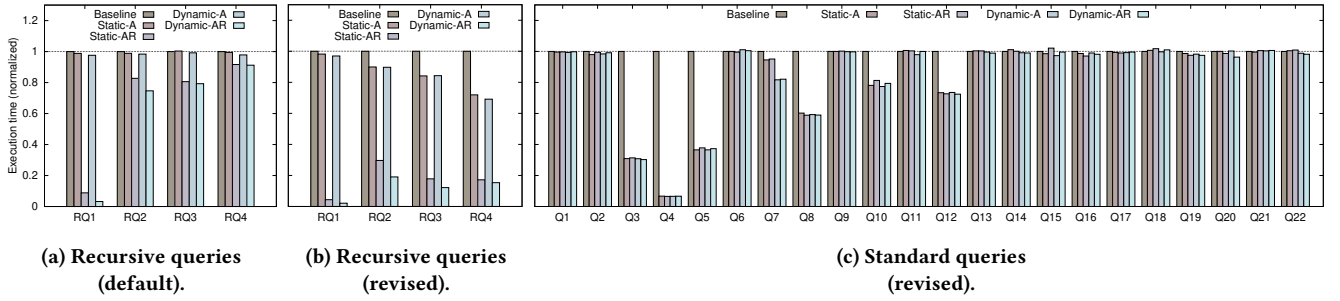


Figure 6: Execution time comparison among different pruning techniques for (a, b) recursive queries (RQ1 to RQ4) and (c) standard queries (Q1 to Q22) on the TPC-H dataset (scale factor: 1,000). *Dynamic pruning of recursive joins (Dynamic-AR) achieves significant speedups for many recursive queries and even for some standard queries on TPC-H. Note that (b) and (c) assume the revised dataset (having a revised order for ORDERKEY to represent the business semantics), whereas (a) assumes the original dataset.*

a scale factor of 120 production days and 30 production lines (approx. 1,036 GB), respectively. Similarly, records of major fact tables (e.g., ORDERS and LINEITEM for TPC-H) were split into different chunks by one day of their date attributes. All the relations were organized in the columnar format [66]. The summary information (implemented as a range index [61, 89, 113]) was set to reserve a value range on each of the primary and foreign key attributes and the major attributes appearing in query predicates in a relation assigned for every chunk.

We executed each test query for each test case and measured its performance metrics such as execution time and a read mount. This paper reports an average value of five trials.

5.1.2 Test machine. We conducted all the experiments on a modern enterprise machine. The machine was composed of Hitachi

HA8000/DL380 Gen10, a database server (having 48 2.4GHz Intel Xeon cores, 768GB memory, and a mirrored pair of two 960GB SSDs) and Hitachi Virtual Storage Platform E790, an external storage subsystem (having 24 7.6TB NVMe SSDs plus a RAID controller). The database server and the external storage subsystem were connected via two 32Gbps Fibre Channel cables. The database server ran Red-Hat Linux 8.3 and HADB 05-07/A, in which the query execution engine was revised to be able to execute a given query in any of the five test cases. The database was built in a storage volume organized by all the SSDs enclosed in the external storage subsystem in RAID-5, whereas the operating system and the other files were stored in an internal storage volume of the database server. The database page size (table/index), the extent size, the buffer size, and the intra-query execution parallelism [30, 105] were set to 32KB/8KB, 4MB, 41GB, and 48 threads, respectively.

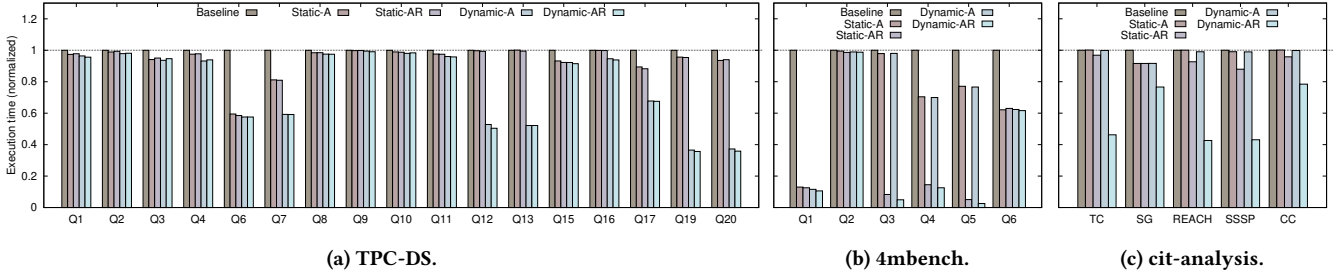


Figure 7: Execution time comparison among different pruning techniques for (a) TPC-DS (scale factor: 1,000), (b) 4mbench (scale factor: 120 production days and 30 production lines) and (c) citation network analysis (cit-analysis) on the cit-Patents dataset. *Dynamic pruning of recursive joins (Dynamic-AR) achieves significant speedups: (a) more than 20% speedups for seven out of 18 standard queries on TPC-DS, (b) up to 39.5 times for recursive queries (Q3, Q4 and Q5), and up to 9.5 times for non-recursive queries on 4mbench, and (c) 27.5% to 134.6% for recursive queries on cit-analysis.*

5.2 A case of recursive join query (RQ1 on TPC-H)

First of all, let us present a case of a test query (RQ1) containing a recursive join operator that works on the standard TPC-H schema and dataset. We compared the execution performance of dynamic pruning with the other pruning techniques for RQ1.

Figure 4 shows that dynamic pruning of recursive joins (**Dynamic-AR**) speeded up the query execution of RQ1 by 30.8 times than the baseline case (**Baseline**) and 2.72 times even than the state-of-the-art technique (static pruning, **Static-AR**).

Figure 4(a) comparatively summarizes the execution time taken for different pruning techniques. The left-most bar indicates **Baseline**, which took 83.2 seconds for the query execution. The next two bars denote static pruning cases, **Static-A** (generating a pruning filter merely on the anchor step at compile time) and **Static-AR** (generating pruning filters on the anchor and recursive steps); the former case reduced the execution time negligibly (by 0.96 seconds), whereas the latter case reduced the execution time significantly (by 74.9 seconds). The last two bars denote dynamic pruning cases, **Dynamic-A** (dynamically generating a pruning filter merely on the anchor step) and **Dynamic-AR** (dynamically generating pruning filters on the anchor and recursive steps); the former case reduced the execution time slightly (by 2.00 seconds), whereas the latter case successfully reduced the execution time more significantly (by 80.5 seconds). Similarly, Figure 4(b) summarizes the speedup ratio of each pruning technique; the speedup ratio was normalized to **Baseline**. This chart shows that **Static-A** and **Dynamic-A** gained negligible or slight speedups; however, **Static-AR** improved the query execution significantly (by 11.3 times), and **Dynamic-AR** offered a further significant speedup (by 30.8 times over **Baseline** and 2.72 times over **Static-AR**).

To investigate the pruning effect on the internal query execution behavior, we measured read requests issued for the storage volume while the query was being processed. Figure 5 shows that dynamic pruning of recursive joins (**Dynamic-AR**) reduced data reads necessary for the query execution of RQ1 by 52.2 times than the baseline case (**Baseline**) and 3.52 times even than the state-of-the-art technique (static pruning, **Static-AR**). Figure 5(a) comparatively summarizes the amount of data reads from storage for

every step of the recursive join processing for different pruning techniques. Obviously, the test query conducted twelve recursive parts following an anchor step; note that the read amount of each step is stacked in the bar chart. The left-most bar indicates that **Baseline** invoked almost the same amount of reads for each recursive step after invoking a smaller amount of reads for the anchor step. The next bar shows that **Static-A** worked very similarly even though it reduced data reads for the anchor step slightly. The center bar indicates that **Static-AR** successfully reduced data reads for each recursive step significantly, but the reduction becomes smaller at later recursive steps. Again, the next bar shows that **Dynamic-A** behaved similarly to **Static-A**. Finally, the last bar indicates that **Dynamic-AR** further significantly reduced data reads for each recursive step, and the reduction successfully continued at later recursive steps.

These observations are supported by Figure 5(b-f), each illustrating an IO trace of data read commands for retrieving ORDERS records for a separate pruning case; the X-axis and Y-axis denote elapsed time and IO address, respectively, and each point indicates a separate read command. Figure 5(b) shows that **Baseline** performed a similar scan on ORDERS for each recursive step. Figure 5(d) indicates that **Static-AR** successfully reduced data reads on ORDERS by exploiting the pruning filter; unfortunately, the pruning effect degraded at later recursive steps, seemingly because the hint was generated only at compile time. In contrast, Figure 5(f) indicates that **Dynamic-AR** worked more successfully to reduce data reads, and the pruning effect was valid even for later recursive steps. This was seemingly caused by the dynamic nature of **Dynamic-AR**; it dynamically generated the pruning filter, thus being able to offer an effective pruning filter for a later recursive step in consideration of its previous part.

In summary, the experimental observation indicates that **Dynamic-AR** speeded up the test recursive join query significantly, and it outperformed even **Static-AR**. **Dynamic-AR** dynamically generated the pruning filter so that it could offer effective pruning for later recursive steps, whereas **Static-AR** generated the pruning filter only at compile time so that the pruning effect could degrade at later steps.

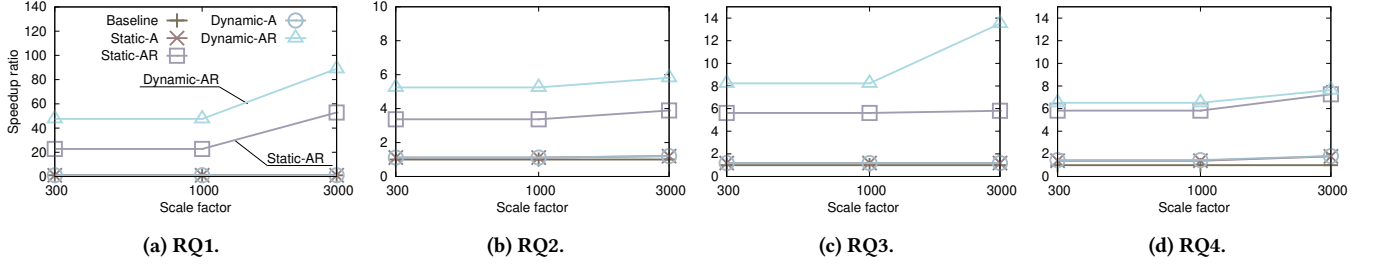


Figure 8: (a-d) Speedup ratios by different pruning techniques for different dataset scales (scale factors: 300 to 3,000) for recursive queries (RQ1 to RQ4) on TPC-H. *Dynamic-AR consistently performs better than the other pruning techniques, and larger datasets allows higher speedups for several queries.*

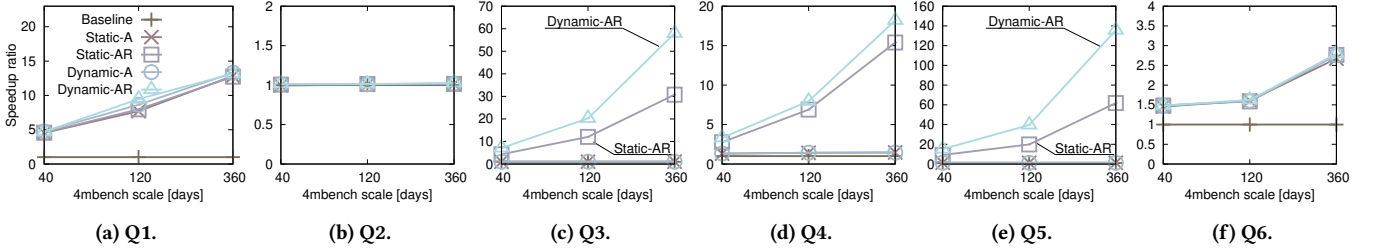


Figure 9: (a-f) Speedup ratios by different pruning techniques for different dataset scales (40 to 360 production days) for test queries (Q1 to Q6) on 4mbench. *Dynamic-AR consistently performs better than the other pruning techniques, and larger datasets allows higher speedups for many queries.*

The remaining part of this section will present our evaluation in a variety of test configurations to clarify the coverage and significance of the dynamic pruning technique.

5.3 Four business benchmark scenarios

TPC-H. Next, we present the performance test that we conducted for all of the additional recursive queries (RQ1 to RQ4) and the original standard queries (Q1 to Q22) on the TPC-H dataset in the same configuration as presented in the previous subsection.

Figure 6 shows that dynamic pruning of recursive joins (**Dynamic-AR**) achieved significant speedups for many recursive queries and even for some standard queries on TPC-H and performed better than the other pruning techniques.

Figure 6(a) summarizes the execution time of different pruning techniques for the four recursive join queries (RQ1 to RQ4). Note that the execution time is normalized based on the baseline case (**Baseline**) for each query. The overall performance property was similar among the queries, although the improvement significance varied. The pruning techniques limited to the anchor part (**Static-A** and **Dynamic-A**) only yielded negligible performance effects. Static pruning (**Static-AR**) offered a significant reduction of execution time by 8.35% to 91.2% from **Baseline**, indicating 1.09 to 11.3 times speedups. Dynamic pruning (**Dynamic-AR**) consistently outperforms **Static-AR**, reducing execution time by 8.79% to 96.8% from **Baseline**, indicating 1.10 to 30.8 times speedups.

Table 2 summarizes the CPU time consumed by **Dynamic-AR** for runtime filter generation and evaluation, measured with Linux profiling tools. This result shows that our concise filter design

successfully works to offer only negligibly small overheads, even though the filter is intensively updated and evaluated at runtime.

In addition, we tested the original standard queries (Q1 to Q22) containing no recursive joins as well. As a result, any pruning techniques yielded no significant performance improvement; specifically, the 95% confidence interval of the observed speedup ratios was $[0.0\%, 1.2\%]$ by **Static-AR** and $[-0.8\%, 0.2\%]$ by **Dynamic-AR** (not presented in the figure), where a negative value indicates performance degradation. The pruning was hardly effective for these queries. The penalty induced by **Dynamic-AR** was seemingly caused by the runtime mechanism of filter generation, but it was acceptably small. We also investigated the cause of this poor result. Although the business scenario of TPC-H presumes that the ORDERKEY and ORDERDATE values are correlated by nature, the standard dataset generator **DBGEN** provided by TPC [93] does not implement this assumption, but it generates both values in an independent order³, thus providing unrealistically ordered records for ORDERS. To this end, we revised the dataset to make it represent the business semantics more accurately and conducted the same experiment⁴. Figures 6(b) and (c) summarize the execution time of different pruning techniques for the recursive queries and the standard queries; **Dynamic-AR** achieved significant (up to 47.6 times) speedups for all the recursive queries, and interestingly, it

³The major implementation practice presumes that, every time a new order is registered into the database, the sequencer generates a unique value by incrementing its previously generated value and assigns the new value to ORDERKEY. Thus, later orders (having larger ORDERDATE values) have larger ORDERKEY values.

⁴We renumbered ORDERKEY in the order of ORDERDATE. Reference [65] employed a similar configuration by appending yet another key attribute to replace ORDERKEY.

Table 2: CPU time (95% confidence interval) consumed by dynamic pruning for filter generation and evaluation. (d) and (r) indicate (default) and (revised), respectively, in Figure 6.

Database	Queries	Generation	Evaluation
TPC-H (d)	RQ1-4	0.0003% - 0.0028%	0.0035% - 0.0503%
TPC-H (r)	RQ1-4	0.0013% - 0.0073%	0.0056% - 0.0507%
TPC-H (r)	Q1-22	0.1526% - 0.4659%	0.0008% - 0.0023%
TPC-DS	Q1-20	0.0344% - 0.5432%	0.0363% - 0.4865%
4mbench	Q1-6	0.0942% - 0.5924%	0.0144% - 0.0569%
cit-analysis	all	0.0202% - 0.0744%	0.0402% - 0.0830%

succeeded for seven out of 22 standard queries to achieve more than 20% speedups (up to 15.1 times). This result implies that **Dynamic-AR** potentially works for such queries containing normal joins as well as recursive ones. Table 2 shows that it only yielded very small processing overheads for standard queries, implying that **Dynamic-AR** does not slow down non-effective queries.

TPC-DS. Next, we present the performance experiment with TPC-DS, another decision-support benchmark. We tested 17 standard queries (Q1 to Q4, Q6 to Q13, Q15 to Q17, and Q19 to Q20)⁵. We observed a similar result on TPC-DS with TPC-H. Figure 7(a) shows that **Dynamic-AR** effectively worked for TPC-DS. **Dynamic-AR** succeeded for seven out of 17 standard queries to achieve more than 20% speedups (up to 2.8 times), even though they only contained normal joins rather than recursive ones. Filter processing overheads were negligible (Table 2).

4mbench. We also present the performance test that we conducted similarly with 4mbench, yet another benchmark simulating manufacturing business analytics. Figure 7(b) shows that **Dynamic-AR** consistently offered significantly (up to 39.5 times) better performance than the other pruning techniques on recursive queries (Q3, Q4 and Q5) and even on non-recursive join queries (Q1 and Q6). It always performed better even than **Static-AR** but its performance improvement varies with different recursive queries. One exception was that any pruning techniques yielded no significant performance improvement or degradation on a specific query (Q2).

Citation network analysis (cit-analysis). Finally, we present the test of citation network analysis, where five recursive queries were run on the cit-Patents dataset. Figure 7(c) shows that **Dynamic-AR** consistently performed effectively, offering significant speedups (up to 134.6% faster), even though these recursive queries were originally designed for DataLog but converted to relational database.

In summary, our experiments clarifies that **Dynamic-AR** substantially speeds up recursive join queries and non-recursive queries having deeply nested joins, performing better even than **Static-AR**. The additional processing induced by **Dynamic-AR** is insignificant; an acceptably small performance penalty is observed for the queries where the pruning is unsuccessful.

5.4 Dataset scales

We present the experimental study on the dataset scalability. We conducted the same performance test as presented in the previous

⁵We omitted Q5, Q14, and Q18 from the performance test because our current implementation did not support the pruning capability for ROLLUP or CUBE.

subsection while changing dataset scales. Specifically, we generated datasets with different scale factors for TPC-H and different production days for 4mbench, respectively. The remaining part of the paper only highlights the recursive queries (RQ1 to RQ4) for TPC-H⁶ and all the test queries (Q1 to Q6) for 4mbench due to the space limitation. Figure 8 presents the same performance property for different dataset scales on TPC-H as we observed in the previous experiments; dynamic pruning of recursive joins (**Dynamic-AR**) offered better performance than the other pruning techniques, even though the performance improvements varied with different queries and different dataset scales. Particularly, Figures 8(a) and (c) show the execution of RQ1 and RQ3, indicating that **Dynamic-AR** achieved significantly larger speedups as the dataset scale increased. Our investigation confirmed that larger dataset scales yielded lower selectivity ratios for these queries, thus making the pruning more effective.

We observed similar performance properties on 4mbench. Figure 9 shows that dynamic pruning of recursive joins (**Dynamic-AR**) offered better performance than the others, and the speedup increased as the dataset scale grew for most of the test queries.

5.5 Chunk granularity

We present the experimental study on the chunk granularity sensitivity. We conducted the same performance test while changing chunk granularity values from one day. Specifically, we organized the database with different chunk granularity values for TPC-H and 4mbench, and then tested the query performance.

We basically observed that the choice of chunk granularity values did not significantly affect the performance improvement of any pruning techniques for TPC-H (not shown in the figure) and 4mbench (shown in Figure 10), and dynamic pruning (**Dynamic-AR**) consistently performed the best. Let us deeply dive into three minor exceptions for Q3, Q4 and Q5 on 4mbench; the speedup achieved by static pruning (**Static-AR**) degraded as the chunk granularity increased. This was seemingly because larger chunk granularity made each chunk cover a wider range of key values, and the filter prediction generated at compile time became less accurate. Despite these minor cases, **Dynamic-AR** offered better performance than the others for all the test queries.

5.6 Dataset skewness

Finally, we present the experimental study on the dataset skewness sensitivity. We generated datasets with different skewness factors for TPC-H, and then tested the query performance. The standard dataset generator DBGEN provided by TPC [93] does not have the capability of generating a skewed dataset. Instead, we utilized a third-party generator [108] with the capability of generating a dataset according to Zipf's law with a skewness factor. We chose this generator since it was utilized for other academic studies [25, 71, 74, 99, 104, 109].

Figure 11 summarizes our observation. The x-axis of each chart denotes the skewness factor; 0 indicates the uniform distribution and larger values indicate higher skewness. We again observed that

⁶Hereafter, this paper presents only the result obtained with the revised dataset (introduced in Section 5.3) rather than the original dataset for TPC-H. **Dynamic-AR** outperformed the other pruning techniques on the original dataset similarly, but its speedup ratio varied with queries and configurations.

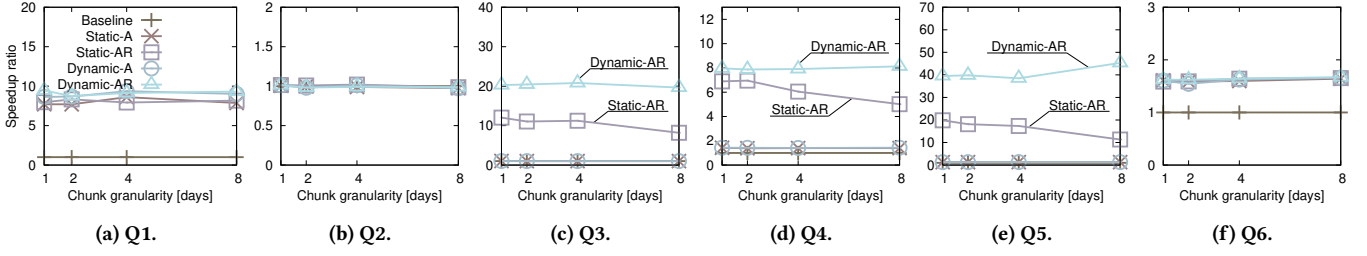


Figure 10: (a-f) Speedup ratios by different pruning techniques for different chunk granularity values (1 to 8 days) for test queries (Q1 to Q6) on the 4mbench dataset (120 production days). The variation of chunk granularity does not significantly affect the performance improvement enabled by Dynamic-AR.

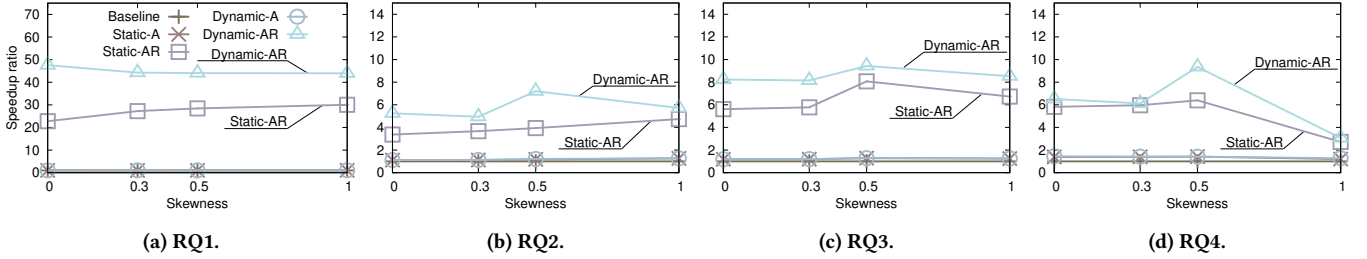


Figure 11: (a-d) Speedup ratios by different pruning techniques for different skewness factors for recursive queries (RQ1 to RQ4) on the TPC-H dataset (scale factor: 1,000). Dynamic-AR performs better than the other pruning techniques on differently skewed datasets, even though the skewness occasionally changes the query processing behavior, affecting the speedup ratio.

dynamic pruning (**Dynamic-AR**) offered better performance than the others, although the dataset skewness affected the speedup ratio. Our investigation indicated that, as the dataset skewness changed, the query processing behavior occasionally changed. For example, the execution of RQ2 needed four recursive steps on the uniform dataset, but it only needed two on the moderately skewed datasets (skew factor: 0.5). Such behavioral changes affected the pruning effect and then the speedup ratio.

Summary. We have experimentally clarified that (1) dynamic pruning performs up to 135.6 times faster than the baseline algorithm, and it also outperforms the state-of-the-art static pruning approach (by up to 3.99 times) for a variety of query and dataset configurations; (2) the relative performance benefit comes from its dynamic property of generating the filter to offer effective pruning for deeply nested joins; and (3) the concise filter design successfully offers only negligibly small performance overheads for intensive runtime filter manipulation, not being harmful to ineffective queries.

6 Conclusion

This paper has explored dynamic pruning for recursive joins. Dynamic pruning exploits the runtime information for building the pruning filter; a separate filter is allowed to be made based on the result of each recursive step, and it helps to significantly reduce the amount of relation records to be accessed at the next recursive step. Hence, dynamic pruning potentially improves query performance, particularly, of recursive join queries having deeply nested joins. We have presented the intensive and extensive experiments

that we have conducted with the commercial prototype to clarify the performance benefit of dynamic pruning with four business benchmark workloads. Our observation has demonstrated that dynamic pruning significantly speeds up recursive join queries and some non-recursive ones; it works up to 135.6 times faster than the baseline case and 3.99 times faster even than the state-of-the-art pruning approach. In addition, it is not harmful to queries to which the pruning is not effective, since it only yielded negligible small processing penalties for runtime filter manipulation.

Future work includes the sophistication of the data structure for the pruning filter. Our current prototype employs a single range in the pruning filter, keeping only the maximum and minimum values for each key attribute. Although this strategy experimentally demonstrates successful pruning effects, there are several cases where we can achieve more pruning effects by extending the pruning filter [110]. On the other hand, complications of the filter design may cause other side effects such as processing overheads. The tradeoff should be carefully studied. In addition, this paper assumes that the join algorithm is static in the recursion, but it may change [28, 38]. The extension to such cases is another open problem.

Acknowledgments

This work has been in part supported by “Big Data Value Co-creation Platform Engineering” social cooperation program at UTokyo-IIS with Hitachi and Cross-ministerial Strategic Innovation Promotion Program on Integrated Health Care System.

References

- [1] Ali Afrozeh, Bogdan Ghit, and Greg Rahn. 2020. Faster SQL Queries on Delta Lake with Dynamic File Pruning. <https://www.databricks.com/blog/2020/04/30/faster-sql-queries-on-delta-lake-with-dynamic-file-pruning.html>
- [2] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable summaries. In *Proc. PODS*. ACM, 23–34.
- [3] Rakesh Agrawal and H. V. Jagadish. 1987. Direct Algorithms for Computing the Transitive Closure of Database Relations. In *Proc. VLDB*. 255–266.
- [4] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.
- [5] Aamir Junaid Ahmad and Sabina Priyadarshini. 2020. Design and development of a friendly user interface for building social network traceability system. *Soc. Netw. Anal. Min.* 10, 1 (2020), 59.
- [6] Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. 2022. Optimizing Differentially-Maintained Recursive Queries on Dynamic Graphs. *Proc. VLDB Endow.* 15, 11 (2022), 3186–3198.
- [7] Pauline Anthonysamy, Phil Greenwood, and Awais Rashid. 2014. A Method for Analysing Traceability between Privacy Policies and Privacy Controls of Online Social Networks. In *Privacy Technologies and Policy*, Bart Preneel and Demosthenes Ikononou (Eds.). 187–202.
- [8] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parghas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proc. SIGMOD*. 2205–2217.
- [9] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. PODS*. ACM, 1–15.
- [10] François Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. SIGMOD*. 16–52.
- [11] Catriel Beeri and Raghu Ramakrishnan. 1987. On the Power of Magic. In *Proc. PODS*. ACM, 269–284.
- [12] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.
- [13] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *Proc. SIGMOD*. 1773–1786.
- [14] Julio Garrido Campos and Luis Rodriguez Miguez. 2009. Manufacturing traceability data management in the supply chain. *Int. J. Inf. Technol. Manag.* 8, 3 (2009), 321–339.
- [15] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276.
- [16] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. 2011. Tenzing A SQL Implementation On The MapReduce Framework. *Proc. VLDB Endow.* 4, 12 (2011), 1318–1327.
- [17] Kuan-Ta Chen and Li-Wen Hong. 2007. User identification based on game-play activity patterns. In *Proc. NETGAMES*. 7–12.
- [18] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1997. On Applying Hash Filters to Improving the Execution of Multi-Join Queries. *VLDB J.* 6, 2 (1997), 121–131.
- [19] Songting Chen. 2010. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *Proc. VLDB Endow.* 3, 2 (2010), 1459–1468.
- [20] Latha S. Colby. 1989. A Recursive Algebra and Query Optimization for Nested Relations. In *Proc. SIGMOD*. 273–283.
- [21] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proc. SIGMOD*. 215–226.
- [22] Edsger W. Dijkstra. 1960. Recursive Programming. In *Numerische Mathematik*. Vol. 2. 312–318.
- [23] Guozhu Dong and Jianwen Su. 2000. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Rec.* 29, 1 (2000), 44–51.
- [24] Wenbin Fang, Barton P. Miller, and James A. Kupsch. 2012. Automated tracing and visualization of software security structure and properties. In *Proc. VizSec*. 9–16.
- [25] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proc. SIGMOD*. 31–46.
- [26] David Fitzjarrell. 2014. Oracle Bloom Filters. <https://www.databasejournal.com/oracle/oracle-bloom-filters/>
- [27] Sofoklis Floratos, Ahmad Ghazal, Jason Sun, Jianjun Chen, and Xiaodong Zhang. 2021. DBSpinner: Making a Case for Iterative Processing in Databases. In *Proc. ICDE*. 2399–2410.
- [28] Ahmad Ghazal, Dawit Yimam Seid, Alain Crolotte, and Mohammed Al-Kateb. 2012. Adaptive optimizations of recursive queries in teradata. In *Proc. SIGMOD*. 851–860.
- [29] Kazuo Goda, Yuto Hayamizu, Norifumi Nishikawa, and Shinji Fujiwara. 2022. 4mbench: Performance Benchmark of Manufacturing Business Database. In *Proc. TPCTC*. 94–109.
- [30] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada, and Masaru Kitsuregawa. 2020. Out-of-order Execution of Database Queries. *Proc. VLDB Endow.* 13, 12 (2020), 3489–3501.
- [31] Goetz Graefe and Harumi A. Kuno. 2010. Fast Loads and Queries. *Trans. Large Scale Data Knowl. Centered Syst.* 2 (2010), 31–72.
- [32] Liyang Gu and Adele Lu Jia. 2018. Player Activity and Popularity in Online Social Games and their Implications for Player Retention. In *Proc. NETGAMES*. 1–6.
- [33] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proc. SIGMOD*. 559–573.
- [34] Ravindra Guravannavar and S. Sudarshan. 2008. Rewriting procedures for batched bindings. *Proc. VLDB Endow.* 1, 1 (2008), 1107–1123.
- [35] Jiawei Han and Lawrence J. Henschen. 1987. Handling Redundancy in the Processing of Recursive Database Queries. In *Proc. SIGMOD*. 73–81.
- [36] Emil Riis Hansen, Thomas Dyhre Nielsen, Thomas Mulvad, Mads Nibe Stausholm, Tomer Sagi, and Katja Hose. 2023. Patient Event Sequences for Predicting Hospitalization Length of Stay. In *Proc. AIME*. 51–56.
- [37] Yuto Hayamizu, Ryoji Kawamichi, Kazuo Goda, and Masaru Kitsuregawa. 2018. Benchmarking and Performance Analysis of Event Sequence Queries on Relational Database. In *Proc. TPCTC*. 110–125.
- [38] Anna Herlihy, Guillaume Martres, Anastasia Ailamaki, and Martin Odersky. 2024. Adaptive Recursive Query Optimization. In *Proc. ICDE*. 368–381.
- [39] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *Proc. SIGMOD*. 723–735.
- [40] Hitachi Ltd. 2011. Manuals - Middleware - Hitachi Advanced Database. https://itpfdoc.hitachi.co.jp/Pages/region/document_list/manuals/en/hadb.html
- [41] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *Proc. ICDE*. 774–783.
- [42] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaida. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *Proc. SIGMOD*. 681–697.
- [43] Henry S. Warren Jr. 1975. A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations. *Commun. ACM* 18, 4 (1975), 218–220.
- [44] Poongodi K and Dhananjay Kumar. 2023. Mining Frequent Serial Positioning Episode Rules with Forward and Backward Search Technique from Event Sequences. *Comput. J.* 66, 7 (2023), 1622–1643.
- [45] Wooseok Kang, Byoungso Son, and Kihong Heo. 2022. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. In *Proc. CCS*. 1695–1708.
- [46] Amanda Kinchla, Nicole Richards, and Lori Pivarnik. 2016. Food Safety Plan for Chocolate Chip Cookie Teaching Example. <https://ag.umass.edu/sites/ag.umass.edu/files/cookiefoodsafetyplan.pdf>, accessed on June 19, 2022.
- [47] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. 2020. Cuckoo Index: A Lightweight Secondary Index Structure. *Proc. VLDB Endow.* 13, 13 (2020), 3559–3572.
- [48] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovitsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. CIDR*.
- [49] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469.
- [50] Marlene Kuhn, Felix Funk, Guanlai Zhang, and Jörg Franke. 2021. Blockchain-based application for the traceability of complex assembly structures. *J. Manuf. Syst.* 59 (2021), 617–630.
- [51] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Proc. SIGMOD*. 5–17.
- [52] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proc. SIGMOD*. 311–326.
- [53] Fátima Leal, Adriana E. Chis, Simon Caton, Horacio González-Vélez, Juan M. García-Gómez, Marta Durá, Angel Sánchez-García, Carlos Sáez, Anthony Karageorgos, Vassilis C. Gerogiannis, Apostolos Xenakis, Efthymios N. Lallas,

- Theodoros Ntounas, Eleni Vasileiou, Georgios Mountzouris, Barbara Otti, Penelope Pucci, Rossano Papini, David Cerrai, and Mariola Mier. 2021. Smart Pharmaceutical Manufacturing: Ensuring End-to-End Traceability and Data Integrity in Medicine Production. *Big Data Res.* 24 (2021), 100172.
- [54] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [55] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Proc. SIGMOD*. 334–346.
- [56] Yanan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2 (2023), 178:1–178:26.
- [57] Hongjun Lu and Michael J. Carey. 1985. Some Experimental Results on Distributed Join Algorithms in a Local Network. In *Proc. VLDB*. 292–304.
- [58] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. VLDB*. 149–159.
- [59] Ministry of Health, British Columbia. 2017. Food Safety Plan Workbook. https://www2.gov.bc.ca/assets/gov/health/keeping-bc-healthy-safe/food-safety-security/food_safety_plan_workbook_sept6_2017.pdf, accessed on June 19, 2022.
- [60] Grace M. Mirsky and María de la Salud Guillem Sánchez. 2016. Analysis of the Clinical Utility of Algorithms in the 2009 PhysioNet/Computing in Cardiology Challenge for the Prediction of Acute Hypotensive Episodes. In *Proc. CinC*.
- [61] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proc. VLDB*. 476–487.
- [62] James K. Mullin. 1990. Optimal Semijoins for Distributed Database Systems. *IEEE Trans. Software Eng.* 16, 5 (1990), 558–560.
- [63] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-sets in a Relational Database System. In *Proc. SIGMOD*. 103–114.
- [64] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *Proc. SIGMOD*. 627–640.
- [65] Anisoara Nica, Reza Sherkat, Mihnea Andrei, Xun Chen, Martin Heide, Christian Bensberg, and Heiko Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *Proc. VLDB Endow.* 10, 12 (2017), 1658–1669.
- [66] Norifumi Nishikawa, Shinji Fujiwara, Yuto Hayamizu, and Kazuo Goda. 2023. Physical Database Design for Manufacturing Business Analytics. In *Proc. IEEE BigData*. 1793–1802.
- [67] Makoto Onizuka, Hiroyuki Kato, Soichiro Hidaka, Keisuke Nakano, and Zhenjiang Hu. 2013. Optimization for iterative queries on MapReduce. *Proc. VLDB Endow.* 7, 4 (2013), 241–252.
- [68] Carlos Ordonez. 2005. Optimizing recursive queries in SQL. In *Proc. SIGMOD*. 834–839.
- [69] Carlos Ordonez. 2010. Optimization of Linear Recursive Queries in SQL. *IEEE Trans. Knowl. Data Eng.* 22, 2 (2010), 264–277.
- [70] Carlos Ordonez, Achyuth Gurram, and Nirmala Rai. 2014. Recursive Query Evaluation in a Column DBMS to Analyze Large Graphs. In *Proc. DOLAP*. 71–80.
- [71] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265.
- [72] Kisung Park, Hojin Seo, Mostofa Kamal Rasel, Young-Koo Lee, Chanho Jeong, Sung Yeol Lee, Chungmin Lee, and Dong-Hun Lee. 2019. Iterative Query Processing based on Unified Optimization Techniques. In *Proc. SIGMOD*. 54–68.
- [73] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.
- [74] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *Proc. ICDE*. 600–611.
- [75] Thuong-Cang Phan, Anh-Cang Phan, Thi-To-Quyen Tran, and Ngoan-Thanh Trieu. 2019. Efficient Processing of Recursive Joins on Large-Scale Datasets in Spark. In *Proc. ICCSAMA*. 391–402.
- [76] Piotr Przymus, Aleksandra Boniewicz, Marta Burzanska, and Krzysztof Stencel. 2010. Recursive Query Facilities in Relational Databases: A Survey. In *Proc. DTA/BDBT*, Vol. 118. 89–99.
- [77] Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, Divesh Srivastava, and S. Sudarshan. 1993. The CORAL Deductive Database System. In *Proc. SIGMOD*. 544–545.
- [78] Louisa Raschid and Stanley Y. W. Su. 1986. A Parallel Processing Strategy for Evaluating Recursive Queries. In *Proc. VLDB*. 412–419.
- [79] Lucas C. Scabora, Gabriel Spadon, Mirela T. Cazzolato, Daniel S. Kaster, Agma J. M. Traina, José F. Rodrigues Jr., and Caetano Traina Jr. 2021. SHARq: sharing recursive queries in relational databases. In *Proc. SAC*. 336–339.
- [80] Lothar Schmitz. 1983. An improved transitive closure algorithm. *Computing* 30, 4 (1983), 359–371.
- [81] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proc. SIGMOD*. 435–446.
- [82] S. Seshadri and Jeffrey F. Naughton. 1991. On the Expected Size of Recursive Datalog Queries. In *Proc. PODS*. 268–279.
- [83] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proc. SIGMOD*. 1135–1149.
- [84] Lakshmi Kant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the Vertica analytic database: Lessons learned. In *Proc. ICDE*. 1196–1207.
- [85] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. 2001. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *Proc. ICDE*. 575–584.
- [86] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proc. SIGMOD*. 1115–1126.
- [87] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *Proc. VLDB Endow.* 10, 4 (2016), 421–432.
- [88] Jukka Teuhola. 1994. An Efficient Relational Implementation of Recursive Relationships using Path Signatures. In *Proc. ICDE*. 348–355.
- [89] The PostgreSQL Global Development Group. 2016. Chapter 62. BRIN Indexes - PostgreSQL 9.5 Documentation.
- [90] Daniel Ting and Rick Cole. 2021. Conditional Cuckoo Filters. In *Proc. SIGMOD*. 1838–1850.
- [91] Nga Tran, Andrew Lamb, Lakshmi Kant Shrinivas, Sreenath Bodagala, and Jaimin Dave. 2014. The Vertica Query Optimizer: The case for specialized query optimizers. In *Proc. ICDE*. IEEE Computer Society, 1108–1119.
- [92] Transaction Processing Council. 2021. TPC-DS Version 2 and Version 3. <https://www.tpc.org/tpcds/>
- [93] Transaction Processing Council. 2022. TPC-H Version 2 and Version 3, Revision 3.0.1. <https://www.tpc.org/tpch/>
- [94] UTokyo-IIS Data Platform Lab. 2022. 4mbench is a tool for performance benchmark of manufacturing business database. <https://github.com/dbc-utokyo/4mbench>
- [95] UTokyo-IIS Data Platform Lab. 2024. SQL Queries for Evaluating Recursive Join Processing. <https://github.com/dbc-utokyo/recursive-join-queries>
- [96] Moshe Y. Vardi. 1988. Decidability and Undecidability Results for Boundedness of Linear Recursive Queries. In *Proc. PODS*. 341–351.
- [97] Fei Wang, Noah Lee, Jianying Hu, Jimeng Sun, and Shahram Ebadollahi. 2012. Towards heterogeneous temporal clinical event pattern discovery: a convolutional approach. In *Proc. SIGKDD*. 453–461.
- [98] Qiang Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. 2020. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *Proc. SIGMOD*. 2439–2454.
- [99] Yong Wang, Xiaochun Yun, Xi Wang, Shupeng Wang, and Yongshang Wu. 2017. LBFM: Multi-Dimensional Membership Index for Block-Level Data Skipping. In *Proc. ICDE*. 343–351.
- [100] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. In *Proc. SIGMOD*. 79–93.
- [101] Stephen Warshall. 1962. A Theorem on Boolean Matrices. *J. ACM* 9, 1 (1962), 11–12.
- [102] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *Proc. SIGMOD*. 1433–1446.
- [103] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proc. SIGMOD*. 13–24.
- [104] Wenjian Xu, Ziqiang Feng, and Eric Lo. 2016. Fast Multi-Column Sorting in Main-Memory Column-Stores. In *Proc. SIGMOD*. 1263–1278.
- [105] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. 2023. Nested Loops Revisited Again. In *Proc. ICDE*. IEEE, 3708–3717.
- [106] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 136:1–136:28.
- [107] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel R. Madden. 2010. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proc. ICDE*. 657–668.
- [108] YSU Data Lab. 2022. TPC-H-Skew Public: TPC-H benchmark with skew factor enabled. <https://github.com/YSU-Data-Lab/TPC-H-Skew>
- [109] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *Proc. SIGMOD*. 469–480.
- [110] Jia Yu and Mohamed Sarwat. 2016. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *Proc. VLDB Endow.* 10, 4 (2016), 385–396.
- [111] Chao Zhang, Angela Bonifati, Hugo Kapp, Vlad Ioan Haprian, and Jean-Pierre Lozi. 2023. A Reachability Index for Recursive Label-Concatenated Graph Queries. In *Proc. ICDE*. 67–81.

- [112] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Janaki Lahorani, Dmitry Potapov, and Murali Krishna. 2017. Dimensions Based Data Clustering and Zone Maps. *Proc. VLDB Endow.* 10, 12 (2017), 1622–1633.
- [113] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. VLDB.* 723–734.