Extending SQL to Return a Subdatabase

Joris Nix
Saarland University
Saarland Informatics Campus
Saarbrücken, Saarland, Germany
joris.nix@bigdata.uni-saarland.de

Jens Dittrich
Saarland University
Saarland Informatics Campus
Saarbrücken, Saarland, Germany
jens.dittrich@bigdata.uni-saarland.de

Abstract

Every SQL statement is limited to return a single, possibly denormalized table. This approximately 50-year-old design decision has far-reaching consequences. The most apparent problem is the redundancy introduced through denormalization, which can result in long transfer times of query results and high memory usage for materializing intermediate results. Additionally, regardless of their goals, users are forced to fit query computations into one single result, mixing the data retrieval and transformation aspect of SQL. Moreover, both problems violate the principles and core ideas of normal forms.

In this paper, we argue for eliminating the single-table limitation of SQL. We extend SQL's `SELECT` clause by the keyword '`RESULTDB`' to support returning a *result subdatabase*. Our extension has clear semantics, i.e., by annotating any existing SQL statement with the `RESULTDB` keyword, the DBMS returns the tables participating in the query, each restricted to the relevant tuples that occur in the traditional single-table query result. Thus, we *do not* denormalize the query result in any way. Our approach has significant, far-reaching consequences, impacting the querying of hierarchical data, materialized views, and distributed databases, while maintaining backward compatibility. In addition, our proposal paves the way for a long list of exciting future research opportunities.

We propose multiple algorithms to integrate our feature into both closed-source and open-source database systems. For closed-source systems, we provide several SQL-based rewrite methods. In addition, we present an efficient algorithm for cyclic and acyclic join graphs that we integrated into an open-source database system.

We conduct a comprehensive experimental study. Our results show that returning multiple individual result sets can significantly decrease the result set size. Furthermore, our rewrite methods and algorithm introduce minimal overhead and can even outperform single-table execution in certain cases.

CCS Concepts

• **Information systems** → **Structured Query Language**; **Query optimization**.

SIGMOD 2025, June 22–27, 2025, Berlin, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

Keywords

SQL, extension, denormalization, subdatabase, query processing, query optimization

ACM Reference Format:
Joris Nix and Jens Dittrich. 2025. Extending SQL to Return a Subdatabase. In *Proceedings of (SIGMOD 2025).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

With the invention of the relational model by E. F. Codd [12], *database normalization* was already identified as a key principle in database design. In that seminal paper, Codd introduced the first normal form with the aim of enabling the development of a "universal data sublanguage". In his follow-up work [13], Codd went on to define the second and third normal form. These normal forms were designed to reduce data redundancies by eliminating undesirable dependencies between relations, improve maintenance and consistency of the data, enhance the extensibility of databases, and make the relational model more informative to users. Moreover, the logical database schema serves as the common interface for both applications and database developers. Therefore, carefully designing the logical database schema is crucial when creating a database. However, queries involving multiple relations often inherently and inevitably denormalize the underlying data when producing the result set, which contradicts the core principles of normal forms. This implies that significant effort is invested in normalizing data within the database system, yet the normalized state is often neglected once the data is queried using SQL. This occurs, for instance, when passing data to users or creating materialized views.

1.1 Problem Statement

Consider the SQL query in Listing 1. Figure 1 depicts the corresponding database tables with sample data and Figure 2 shows the query result.

Listing 1: SQL statement.

```
SELECT c.name, p.name, p.category
FROM customers AS c, order AS o, products AS p
WHERE c.state = 'NY' AND
      c.id = o.cid AND
      p.id = o.pid
```

When restricted to a single-table query output, computing the relational result shown in Figure 2 has two notable problems.

Problem 1: Relational information redundancy. The result table contains redundancies, with customer names and product information appearing multiple times due to denormalization caused by the underlying join operation. These data values are not only displayed multiple times but are also physically duplicated. In general, the

customers

id	*name*	*state*
0	cust_A	NY
1	cust_B	CA
2	cust_C	NY

order

cid	*pid*
0	1
1	1
2	2
2	1
0	2
1	3

products

id	*name*	*category*
0	smartphone	electronics
1	laptop	electronics
2	shirt	clothing
3	pants	clothing

Figure 1: Database tables with sample data. The gray rows indicate the tuples contributing to the result set in Figure 2.

customers ⋈ order ⋈ products

c.name	*p.name*	*p.category*
cust_A	laptop	electronics
cust_A	shirt	clothing
cust_C	laptop	electronics
cust_C	shirt	clothing

Figure 2: Relational result table. The colors represent attribute values of database entities (tuples) that get duplicated in the result of the query depicted in Listing 1.

larger the result table, the greater the effort to transmit or store these result sets. This is particularly problematic for queries that produce large result sets, as it can lead to significant memory consumption during query processing when these redundancies have to be materialized at some point in the physical execution plan.

Problem 2: Relational information loss. In addition, relational information indicating that duplicated values originate from the same tuple is lost. For instance, there could be two customers named $cust_A$, and distinguishing between them in the result table would require projecting their primary keys. Another issue with the result table is that the concept of a "key" is simply abandoned when processing data in SQL or relational algebra. For instance, in Figure 2, the combination of the customer and product name uniquely identifies each record. However, this information is not evident from the result set alone. SQL discards even more information from the underlying relations: where did a specific attribute in the result table originate from? Does that attribute value correspond to an attribute in one of the base relations, or was it computed? These are central questions in the area of "data provenance" and emphasize that both SQL and relational algebra are not primarily data *retrieval* languages but rather data *transformation* languages. Both languages take a set of base relations as their input and transform them into **a single output relation**. However, considerable information regarding the relationship among the schema, keys, and data from the base relations is lost in the process.

Therefore, in this work, we argue that it is much more natural to return individual, reduced tables, i.e., a subdatabase, instead of a potentially denormalized result.

Definition 1.1 (Result Subdatabase). Let Q be an arbitrary select-project-join query over a set of relations $R = \{R_1, \ldots, R_n\}$ that projects to a set of attributes $A = A_1 \cup \cdots \cup A_m$ where each A_i is a subset of the attributes of relation R_i and $m \leq n$. Let T be the single table result of Q over R. A result subdatabase is defined as:

$$Q_{\text{subdatabase}} := \{\pi_{A_1}(T), \ldots, \pi_{A_m}(T)\}$$

In other words, instead of returning a single table, we return the set of tables whose attributes are part of the projection of the original query, each containing only the tuples that contribute to the overall query result. This concept has a wide range of use cases where it enhances declarative simplicity and may even improve query performance, a selection of which are discussed in the following.

electronics

id	*pid*	*storage*
0	0	64 GB
1	0	32 GB
2	1	128 GB

products

id	*name*	*price*
0	smartphone	900
1	laptop	3500
2	shirt	40
3	pants	120

clothing

id	*pid*	*size*
0	2	L
1	3	XS
2	3	M

Figure 3: Database tables showing *electronics* and *clothing* as subtypes of *products*. The *pid* in both subtypes is a foreign key to the supertype.

1.2 Use Cases

1. Hierarchical Data. Consider the database tables shown in Figure 3, which shows a different way of modeling the products table. Instead of having the category as an attribute as shown in Figure 1, the products table is divided into multiple subtypes, each representing a specific category and containing category-specific information. Let's assume we are interested in all electronic and clothing products priced under 1000 Euros. Listing 2 shows the corresponding query.

Listing 2: Querying hierarchical data.

```
SELECT e.*, c.*
FROM products AS p
     LEFT OUTER JOIN electronics AS e ON p.id = e.pid
     LEFT OUTER JOIN clothing AS c ON p.id = c.pid
WHERE p.price < 1000;
```

Note that, we cannot use `UNION` to compute the desired result due to the different schemas of electronics and clothing. Furthermore, since we must merge the relevant tuples from both subtypes into a single output relation, we are forced to use `OUTER JOIN`s, which introduce `NULL` values as padding. However, with the `RESULTDB` extension, we can compute the same result split into multiple individual output relations, allowing us to eliminate the undesired and redundant `NULL` values. This use case applies more broadly to any scenario in which we need to retrieve data from multiple distinct relations that lack a direct relationship.

2. Views. Materialized views (MVs) are a powerful concept in database management systems, commonly used to precompute specific results and enhance query performance. However, MVs come with drawbacks, the most significant being storage overhead. Materializing query results as views requires physically replicating part of the underlying data, leading to additional storage costs. This issue is exacerbated by the fact that MVs often contain redundancies introduced by data denormalization through joins.

Therefore, applying the idea of materializing only the individual result sets – using the RESULTDB keyword to create the view – offers significant advantages by greatly reducing storage overhead, primarily by eliminating duplicated data. For example, assume we want to create a materialized view for the query given in Listing 1. Instead of materializing the single-table join result with redundancies, as shown in Figure 2, RESULTDB would only materialize the underlying database entities contributing to the query result, as illustrated in gray in Figure 1. Depending on the amount of redundancy in the data, this approach has the potential to considerably reduce storage overhead. The idea of storing only these filtered relations can also be naturally applied to *data provenance*, particularly in the context of view lineage [14]. In that work, the authors propose several algorithms to reconstruct those tuples for a given data item that produce a materialized (aggregation) view. These computed sets of source data items essentially map one-to-one to our reduced base table views and can be used by the proposed algorithms to trace the lineage of data items.

The issue with only storing the filtered base relations is that a **post-join** might be required, i.e., we might have to join the individual tables again. However, this should not be viewed as a disadvantage but rather as an opportunity. On the one hand, if the materialized view contains a high amount of redundancy and the cost of executing the post-join is relatively low, it can be beneficial to send the individual result sets to the client and execute the post-join there, thereby reducing transfer overhead. On the other hand, without fully materializing the join result, we can apply filters and create index structures directly on the filtered base table views, which can be much more efficient than doing so on potentially large materialized views. Furthermore, our experiments (Section 6.4) show that the post-join overhead is in general extremely small.

Another advantage of computing a result subdatabase is that it allows users to conveniently define a completely customized view across multiple tables. With traditional SQL, users are limited to either defining a view for each table individually or combining data from different tables into a single table. Computing a subdatabase can be particularly useful, for instance, when defining a view related to *logical data independence* or *access control*. In addition, it is much more convenient to redefine the view if the requirements or specifications change.

3. Distributed Database Systems. In a distributed setting, it is often advantageous to process as much data as possible locally on a single node. Once processed, the results may need to be sent to another node for further computation, which can lead to significant data transfer overhead. Therefore, computing a result subdatabase locally – rather than a single-table result – can minimize the amount of data that needs to be transmitted, thereby reducing transfer time and potentially decreasing the overall computation time. This concept is similar to the general idea of semi-join reductions in distributed settings [6], which can be seen as a subset of computing a result subdatabase but also applies to broader contexts, such as shipping the result of a query to an application server. In general, any scenario that involves transferring data over a potentially slow network can benefit from producing individual result sets. Naturally, we must consider the trade-off between reducing transfer costs and executing the post-join.

Overall, there are numerous use cases that could benefit from computing multiple individual result sets. The advantages are extensive, including smaller (intermediate) result sets, a more intuitive experience for users, and new opportunities for query optimization.

1.3 Contributions

We extend SQL to allow it to return a result subdatabase, i.e., only the tuples from those relations that are required to compute the query result. In summary, our contributions are as follows:

(1) We introduce a backward-compatible SQL extension, SELECT RESULTDB, which enables SELECT statements in SQL to return a clearly defined subset of a database rather than just a single table. Although the introduction of this new keyword does not extend the expressive power of SQL, it fundamentally alters the underlying semantics of the computed result set. Note that, this work focuses on the *data retrieval* aspect and is therefore limited to select-project-join (SPJ) queries. However, we plan to address data transformation such as grouping and more complex operations such as set difference or anti-joins in future work. (Section 2)
(2) We propose four rewrite algorithms that enable any SQL-92-compliant closed-source database system to support our extension. (Section 3)
(3) We present an efficient native algorithm, that enables query optimizers to compute the result subdatabase efficiently directly inside a database system. We implement our algorithm in mutable [21], an open-source main memory DBMS featuring a state-of-the-art compiling query execution engine. (Section 4)
(4) We conduct an extensive experimental study comparing traditional single-table query processing with our proposed approaches. We evaluate both our rewrite methods and the integration of our algorithm directly into a DBMS. Our results show that multiple individual result sets significantly reduce size, with our methods adding minimal overhead and, in some cases, even outperforming single-table execution. (Section 6)

2 Querying a Database to Return a Subdatabase

We propose to change SQL and relational algebra to return a *subdatabase*. That subdatabase is well-defined: for each relation in a query that is part of the final projection, we return the tuples that contribute to the query result.

2.1 Preliminaries

Let $\mathcal{R}$ be set of all relations and let $\mathcal{Q}$ be the set of all select-project-join queries that project to a set of attributes from its input relations.

Definition 2.1 (A Query Returning a Relation). Let $Q \in \mathcal{Q}$ be a query. We define the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *single-table* (ST) result as follows:

$$Q_{\text{ST}} : \mathcal{Q} \times 2^{\mathcal{R}} \rightarrow \mathcal{R}, (Q, R) \mapsto T$$

Here, T is a relation with the schema from Q's final projection.

2.2 A Query Returning a Subdatabase

Definition 2.2 (A Query Returning a Subdatabase). Let $Q \in \mathcal{Q}$ be a query over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ that

projects to a set of attributes $A = A_1 \cup \cdots \cup A_n$ where each A_i is a subset of the attributes of relation R_i. We define the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *subdatabase* as follows:

$$Q_{\text{RDB}} : \mathcal{Q} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}, (Q, R) \mapsto DB, \text{where}$$
$$DB := \{R'_i \mid R_i \in R \wedge A_i \neq \emptyset\}, \text{with } R'_i := \pi_{A_i}\big(Q_{\text{ST}}(Q, R)\big)$$

In other words, Q_{RDB} returns a *subdatabase*. This subdatabase contains a set of relations $R'_i \subseteq R_i$, where each relation is part of the projections in Q. The subset R'_i is defined as the result of the single-table execution of Q, projected to the attributes A_i.

To avoid confusion, the prefix 'sub' in subdatabase refers to the fact that (1.) Q_{DB} returns a subset of the input relations, and (2.) those relations contain a subset of the tuples from the input relations R. Notice that we assume set semantics of relation algebra, i.e., π returns a duplicate-free set. However, extending this to bag semantics is straightforward: our definitions remain unchanged, with the sole adjustment being that the projection operation must preserve duplicates.

2.3 Relationship-Preserving Subdatabase

Definition 2.3 (Relationship-Preserving Subdatabase). Let $Q \in \mathcal{Q}$ be a query over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ that projects to a set of attributes $A = A_1 \cup \cdots \cup A_n$ where each A_i is a subset of the attributes of relation R_i. Let A_i^J be the subset of attributes of relation R_i that are part of the join predicates in Q. We then define $A'_i := A_i \cup A_i^J$ and the evaluation of Q and its input relations $R \subseteq \mathcal{R}$ that produces a *relationship-preserving subdatabase* as follows:

$$Q_{\text{RDB}_{RP}} : \mathcal{Q} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}, (Q, R) \mapsto DB, \text{where}$$
$$DB := \{R'_i \mid R_i \in R \wedge A'_i \neq \emptyset\}, \text{with } R'_i := \pi_{A'_i}\big(Q_{\text{ST}}(Q, R)\big)$$

In other words, we extend the set of projected attributes A_i by the attributes A_i^J required to compute the joins of Q. In particular, we require a relationship-preserving subdatabase to be able to obtain the original single-table result again. Specifically, we can reconstruct the original single-table result by computing Q_{ST} on the reduced database $Q_{\text{RDB}_{RP}}(Q, R)$:

$$Q_{\text{ST}}(Q, R) = Q_{\text{ST}}\big(Q, Q_{\text{RDB}_{RP}}(Q, R)\big).$$

Note that in some cases it is only necessary to consider a subset of the join predicates to recompute the single-table result, e.g., if a join does not contribute to the reconstruction of the original result.

2.4 Extending SQL: `SELECT RESULTDB`

Given a read-only query $Q \in \mathcal{Q}$ over a set of input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$. We propose to extend the `SELECT` clause of Q to

```
SELECT RESULTDB
```

This statement will return $Q_{\text{RDB}}(Q, R)$ as defined in Definition 2.2. In order to compute the post-join, i.e., to recompute the original single-table result again, we have to return $Q_{\text{RDB}_{RP}}(Q, R)$ as defined in Definition 2.3. Furthermore, `SELECT RESULTDB` does not require full materialization of the returned relations. Results can be returned through pipelines, akin to standard single-table query processing, with each relation mapping to an iterator (or cursor) that uses a pull- or push-based streaming interface.

3 SQL-based Rewrite Methods

In this section, we discuss four SQL-based rewrite methods (RMs) that allow us to implement the query semantics introduced in Section 2. Our rewrite methods can be classified along two dimensions, as depicted in Figure 4.

Dimension 1: SQL semi-join strategy. SQL does not provide a keyword for semi-joins. Hence, we have to write semi-joins implicitly, either through an inner join followed by a `SELECT DISTINCT` on a single relation or through a subquery using the `IN` keyword. Semantically, both rewrites express the same query and return the same result set. Unfortunately, the two rewrites may yield very different plans in many database systems (breaking the strict separation of SQL's declarativeness from query optimization).

Dimension 2: Materialization strategy. Instead of *dynamically* computing join results for the semi-join strategies, we can *prematerialize* or cache data. For example, using MVs we can precompute the entire query result or relevant portions, achieving a trade-off between pre-materialization effort and query processing time.

	`SELECT DISTINCT`	Subquery
Dynamic	RM 1	RM 3
Materialized	RM 2	RM 4

Figure 4: SQL semi-join × materialization landscape.

In the following, we first present the intuition behind each RM, followed by a formal definition using relational algebra. Let $Q \in \mathcal{Q}$ over the input relations $R = \{R_1, \ldots, R_n\} \subseteq \mathcal{R}$ be defined as:

$$Q := \pi_A\Big(\sigma_J\big(\sigma_F(R^\times)\big)\Big),$$

with $A = A_1 \cup \cdots \cup A_n$, where each A_i is a subset of the attributes of the relation R_i, $J = J_1 \wedge \cdots \wedge J_n$ where each J_i is the set of join predicates of R_i, $F = F_1 \wedge \cdots \wedge F_n$ where each F_i is the set of filter predicates of R_i, and $R^\times = R_1 \times \cdots \times R_n$ is the Cartesian product of all relations in R. Furthermore, we illustrate the application of each rewrite method using the query shown in Listing 1.

3.1 RM 1: Dynamic `SELECT DISTINCT`

The first rewrite method transforms a single query into multiple queries where each query projects to the unique attributes of exactly one relation that participates in the query output. Formally, RM 1 rewrites Q into multiple $Q_i := \pi_{A_i}(\sigma_J(\sigma_F(R^\times)))$, $\forall i$ where $A_i \neq \emptyset$. Listing 3 provides an example.

Listing 3: RM using multiple `SELECT DISTINCT`s.

```
1 BEGIN TRANSACTION;
2 SELECT DISTINCT c.name -- Rest of Listing 1
3 SELECT DISTINCT p.name, p.category -- Rest of Listing 1
4 COMMIT;
```

We use the `DISTINCT` keyword to prevent any duplicated entities due to the join and wrap the queries into a transaction to guarantee correct results w.r.t. the current committed state of the database.

While RM 1 is relatively straightforward, the primary disadvantage is that it executes the same queries with slightly different SELECT clauses multiple times, including potentially costly joins.

3.2 RM 2: Materialized SELECT DISTINCT

To proactively circumvent that a potentially expensive join is executed multiple times, RM 2 makes use of MVs. In this RM, we first explicitly create a (temporary and unmaintained) MV *once*. This represents a snapshot as of the time the MV is created. Second, we execute the individual queries against that MV and third, we drop the MV. Formally, RM 2 first creates $Q_{\text{MV}} := Q$ and then creates multiple $Q_i := \pi_{A_i}(Q_{\text{MV}})$, $\forall i$ where $A_i \neq \emptyset$. Listing 4 provides an example.

Listing 4: RM using a MV.

```
CREATE MATERIALIZED VIEW MV AS
SELECT * -- Rest of Listing 1.

SELECT DISTINCT c.name FROM MV;
SELECT DISTINCT p.name, p.category FROM MV;
```

RM 2 has the advantage that it avoids the repeated computation of the same query. However, the disadvantage is that we need to materialize a potentially large query result within the DBMS, which can be particularly costly in a disk-based system. Overall, RM 2 trades upfront costs for materializing the join result versus repeated cost for the computation of the join result.

3.3 RM 3: Dynamic Subquery

The third rewrite method tries to steer the query optimizer into using semi-joins internally by providing a hint using subquery syntax. Formally, RM 3 rewrites Q into multiple Q_i using semi-joins, $\forall i$ where $A_i \neq \emptyset$ as follows:

$$\begin{aligned} Q_i &:= \pi_{A_i}\Big(\sigma_J\big(\sigma_F(R^\times)\big)\Big) \\ &\overset{1.}{=} \pi_{A_i}\Big(\sigma_{F_i}(R_i) \bowtie_{J_i} \big(\sigma_{J\setminus J_i}\big(\sigma_{F\setminus F_i}(R^\times \setminus R_i)\big)\big)\Big) \\ &\overset{2.}{=} \pi_{A_i}\Big(\sigma_{F_i}(R_i) \ltimes_{J_i} \big(\sigma_{J\setminus J_i}\big(\sigma_{F\setminus F_i}(R^\times \setminus R_i)\big)\big)\Big) \end{aligned}$$

Transformation step 1. explicitly performs the join between R_i and the rest of the relations $R \setminus R_i$. In transformation step 2., the application of the left semi-join is equivalent to the join as we only project to the attributes of R_i anyway. Listing 5 provides an example.

Listing 5: RM using dynamic subqueries.

```
BEGIN TRANSACTION;
SELECT DISTINCT c.name -- Query 1
FROM customers AS c
WHERE c.state = 'NY' AND c.id IN -- c.id = o.cid
    (SELECT o.cid FROM order AS o, products AS p
     WHERE o.pid = p.id);
-- Query 2: Analogous for 'products'
COMMIT;
```

Note that the subquery does not necessarily have to contain all other relations but only those required to compute the relevant primary key values, which depends on the specific join graph.

With this rewrite, the query optimizer of a database system like PostgreSQL can efficiently execute a semi-join in certain cases without materializing a potentially large join result. However, the effectiveness of the query optimizer in utilizing a semi-join operator depends on various factors and may not always be guaranteed.

3.4 RM 4: Materialized Subquery

The fourth rewrite method essentially materializes a join index, i.e., for all participating relations, it only projects to their primary keys. Formally, RM 4 first creates $Q_{\text{MV}} := \pi_{A^{\text{PK}}}(\sigma_J(\sigma_F(R^\times)))$, with $A^{\text{PK}} = \{A_i^{\text{PK}} \mid 1 \leq i \leq n \wedge A_i \neq \emptyset\}$, where each A_i^{PK} is the set of primary key attributes of R_i. Afterward, RM 4 creates multiple $Q_i := \pi_{A_i}(R_i \ltimes_{J_i^{\text{PK}}} Q_{\text{MV}})$, $\forall i$ where $A_i \neq \emptyset$. J_i^{PK} denotes the join on the primary key attributes. Listing 6 provides an example.

Listing 6: RM using a MV and multiple subqueries.

```
CREATE MATERIALIZED VIEW MV AS
SELECT DISTINCT c.id, p.id -- Rest of Listing 1.

SELECT DISTINCT c.name  -- Query 1
FROM customers AS c
WHERE c.id IN (SELECT c.id from MV);
-- Query 2: Analogous for 'products'
```

RM 4's rationale is that it typically requires significantly less storage than RM 2 and has the potential to leverage semi-joins internally.

4 ResultDB$_{\text{semi-join}}$ Algorithm

In this section, we present an algorithm that can be integrated into a DBMS to efficiently compute SELECT RESULTDB queries. Our algorithm allows us to fully reduce all relations of a join graph with an arbitrary topology. For this, we first discuss how we can leverage Yannakakis' algorithm for acyclic join graph topologies in Section 4.2. Next, in Section 4.3, we show how to transform cyclic queries into acyclic ones to reuse the algorithm from Section 4.2. Finally, we present our complete algorithm in Section 4.4.

4.1 Preliminaries

The core idea is to efficiently *reduce* each individual relation to the minimal set of tuples that participate in the result set as defined in Definition 2.2. For this, we are going to make use of *semi-joins*. The (left) semi-join between two relations R and S is defined as $\text{R} \ltimes \text{S} = \pi_{[\text{R}]}(\text{R} \bowtie \text{S})$. Similar to previous work [5, 12], we use the term *semi-join reduction* or just *reduction* when performing a semi-join. In particular, we say "R is reduced by S" if we perform $\text{R} \ltimes \text{S}$. Depending on the context, a semi-join reduction can also refer to the reduction of every relation that is part of a query.

As already shown in previous work [5], the shape of the join graph is an essential factor for computing a semi-join reduction. A join graph for some query Q is defined as $JG_Q = (R, J)$, where R is the set of relations and J is the set of joins in Q. We will always assume a connected join graph. In the following, we look separately at both *acyclic* and *cyclic* join graph topologies and discuss how we can algorithmically compute our result subdatabases.

Notion of Acyclicity. Note that *acyclicity* can be defined in various ways, with one common definition being α-acyclicity. Intuitively, it can be defined as follows (see Definition 18.2 of Arenas et al. [4] for a more formal definition):

Definition 4.1 (α-acyclicity). A query Q is acyclic iff there exists an equivalent query Q' whose join graph $JG_{Q'}$ is a tree.

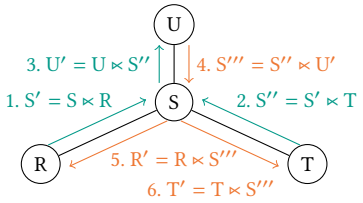


Figure 5: Visualization of a possible semi-join order with U as root. → shows the bottom-up and → the top-down pass.

However, in the scope of this work, we will use a simpler definition solely based on the structure of the join graph:

Definition 4.2 (JG-acyclicity). A query Q is acyclic iff its join graph JG_Q is acyclic.

The problem with these different notions of acyclicity is that a query may be α-acyclic but not JG-acyclic. In this work, we decided to use JG-acyclicity due to the following two reasons. First, checking for JG-acyclicity can be done very efficiently. A join graph is cyclic iff the number of joins is equal to or greater than the number of relations. In contrast, checking for α-acyclicity requires for example the application of the GYO algorithm (see Proposition 18.6 of Arenas et al. [4]), which is computationally more expensive. Second, under α-acyclicity, there can be multiple tree-shaped queries Q' for a given query Q. Deciding which Q' to consider for our algorithm basically represents another enumeration problem.

This presents a trade-off: identifying and constructing acyclic queries under α-acyclicity is computationally more expensive but may save the effort required to transform a cyclic query into an acyclic one (cf. Section 4.3). For the remainder of this paper, we use *(a)cyclicity* to refer specifically to *JG-(a)cyclicity*. However, exploring α-acyclicity will be considered in future work.

4.2 Acyclic Join Graph Topology

The Yannakakis algorithm [40] provides an efficient way for solving acyclic (tree) conjunctive queries, i.e., select-project-join (SPJ) queries that have an acyclic join graph. The algorithm essentially works in three main steps. After choosing an arbitrary root node, we first perform consecutive semi-joins bottom-up from the *leaves* to the *root*. Second, we perform consecutive semi-joins top-down from the *root* to the *leaves*. Lastly, the Yannakakis algorithm joins the reduced relations to obtain a single output relation. The main motivation of this algorithm is to keep intermediate results as small as possible by reducing all relevant relations to their minimal set of tuples that participate in the join before actually joining the relations. Algorithm 1 shows the high level steps of this algorithm.

Algorithm 1 Yannakakis' Algorithm.

(0) Choose an arbitrary node in the join graph as root.
(1) Perform bottom-up semi-joins from leaves to root.
(2) Perform top-down semi-joins from root to leaves.
(3) Compute join result.

Figure 5 shows an acyclic join graph consisting of four relations and visualizes *one possible* semi-join order. In this example, we select U as the root and then compute a breadth-first search order for the edges starting at U. The reversed order now gives us a suitable sequence for the bottom-up semi-joins while we perform the top-down semi-joins in the original order. The semi-joins are performed in the direction of the arrows, i.e., an arrow from R to S represents the semi-join S ⋉ R. Note, that for a specific node with multiple children, the order in which the semi-joins are applied does not matter for correctness but might have an impact on performance.

Algorithm 2 Reduce relations of a join graph using Yannakakis' algorithm.

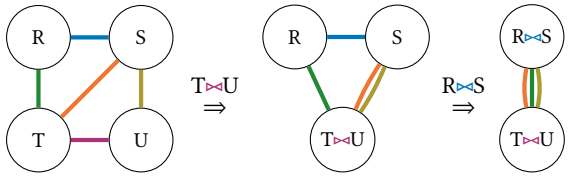
```
1: function REDUCE_RELATIONS(G)                    ▷ G is an acyclic join graph
2:     root = choose_node(G)                       ▷ (0) root node
3:     edges_bfs_order = bfs_edges(G, root)
4:     for join ∈ reversed(edges_bfs_order) do     ▷ (1) bottom-up
5:         semi_join(join.left, join.right)
6:     for join ∈ edges_bfs_order do               ▷ (2) top-down
7:         semi_join(join.right, join.left)
8:     return G                                    ▷ G contains reduced relations
```

Algorithm 2 shows pseudocode for the computation of our result subdatabase based on the fundamental steps in Algorithm 1. In line 2, we choose a root node of our tree-structured join graph (step 0). Instead of randomly selecting a root node, we employ a heuristic that favors relations included in the projections, prioritizing those with higher degrees when multiple such relations exist. This heuristic is based on two key reasons. First, since we only need to reduce relations that we eventually return, choosing a relation in the projections as root can reduce the number of semi-joins needed in the top-down pass. Second, choosing high-degree nodes typically leads to shallower trees, allowing us to perform subsequent semi-joins more effectively. The selection of the root node, which we refer to as the *Root Node Enumeration Problem*, can have a substantial impact on performance. In line 3, we order the edges of the join graph in a breadth-first-search order starting at the root node. Additionally, we assume the join operands are ordered relative to the root node, meaning there are directed edges from the root to the leaves. This is crucial because the semi-join operation is not commutative, and we must ensure that semi-joins are executed in the correct direction. In lines 4-5, we perform the bottom-up semi-joins (step 1), and in lines 6-7, we perform the top-down semi-joins (step 2). In contrast to Yannakakis' algorithm, we no longer need to execute step 3, as the reduced relations already form our desired result database.

The most obvious reason Yannakakis' algorithm is not used in traditional query processing is the uncertainty about whether the overhead of semi-join reduction outweighs the benefits of smaller subsequent join results. However, we need not worry about this since we focus solely on the reduced relations, allowing us to fully leverage the efficiency of Yannakakis' algorithm.

4.3 Cyclic Join Graph Topology

Yannakakis' algorithm is explicitly only defined for acyclic conjunctive queries. Furthermore, Bernstein et al. [5] show that for cyclic queries, semi-joins either cannot be used to *fully reduce* the relations or we need a very long sequence of semi-joins. "Fully reduced" refers to a relation where only the minimal set of tuples needed for the final join result remains. In Figure 5 for example,



(a) JG 1 (b) JG 2 (c) JG 3

Figure 6: Transformation of a cyclic join graph into an acyclic one by folding vertices.

after performing the first semi-join S ⋉ R, the relation S is only partially reduced. Once S is reduced by all neighboring relations, i.e., after the fourth semi-join, it cannot be further reduced anymore. In order to leverage Yannakakis' algorithm, we need to "transform" cyclic queries into tree queries. There is already some work on transforming graphs into a tree structure called *tree decompositions* [34]. We discuss this line of work in more detail in the related work Section 5. However, we decided to employ a more pragmatic and accessible way to get rid of cycles in a join graph.

Algorithm 3 Transform cyclic join graph into acyclic one.

```
1: function FOLD_JOIN_GRAPH(G)
2:     while G.is_cyclic() do                    ▷ #joins >= #relations
3:         x = choose_node(G)
4:         y = choose_neighbor(G, x)
5:         G.replace(x, y, x ⋈ y)                ▷ fold x and y; adjust affected joins
6:     return G                                  ▷ acyclic join graph
```

The general idea is to *fold* relations together such that the cycles in the join graph are resolved. Algorithm 3 shows pseudocode for the construction of an acyclic join graph. At the core, the algorithm consists of two steps. First, we choose a (random) node x in the join graph and one of its neighbors y (lines 3-4). Second, we replace those two nodes x and y with their join result and adjust other affected joins accordingly (line 5). We repeat these steps until our join graph is acyclic (line 2). Since our join graph is undirected and connected, we can easily check for cycles by comparing the number of joins to the number of relations. If the number of joins is equal to or greater than the number of relations, the join graph is cyclic.

Figure 6 shows an example transformation of a cyclic join graph into an acyclic one by folding multiple vertices. Our initial join graph (JG 1) has multiple cycles (e.g. R–S–U–T–R or R–S–T–R). In the first step, we choose the nodes T and U and replace them in the join graph with their join result T ⋈ U. Note that we now have a conjunctive join predicate between S and T ⋈ U which is visualized by having multiple edges in their respective color between two nodes. After this transformation, we still have a cyclic join graph (JG 2). Therefore, we repeat the process and join the nodes R and S. The final join graph (JG 3) consists of two nodes connected by a single join containing three join predicates.

Note, that this is just *one* possible outcome because we did not specify how to choose either node x or y in Algorithm 3. Choosing nodes S and T initially would have yielded an acyclic join graph already after folding the first two nodes. The choice of folds is very likely to have a significant impact on the performance of the algorithm. This opens up a whole new optimization problem that we coin the *Tree Folding Enumeration Problem*. Exploring this problem goes beyond the scope of this paper, and we will investigate it as part of future work. However, instead of just randomly picking the two nodes x and y, our implementation heuristically chooses the nodes with the highest degree. The rationale for this is that nodes with many join partners are more likely to be part of a cycle and with that, we might need fewer folds to reach an acyclic state.

Lemma 4.3. *The folding process does not alter the join result and eventually results in an acyclic join graph.*

Proof. Let $Q \in \mathcal{Q}$ be a cyclic query and $JG_Q = (R, J)$ be the corresponding connected join graph, where $R = \{R_1, R_2, \ldots, R_n\}$ is the set of relations and J is the set of joins in Q. Joining two arbitrary but connected relations R_i and R_j and replacing them with the join $R_i \bowtie R_j = R_{ij}$ in Q does not change the overall join result. This follows directly from the associativity of the join operation.

A connected, undirected join graph is acyclic iff the number of joins $|J|$ is equal to the number of relations $|R|$ minus 1, i.e., $|J| = |R| - 1$. Each folding step reduces the number of relations in the join graph by *exactly one* by merging two nodes. However, folding two nodes reduces the number of joins in the join graph by *at least one*, depending on the structure of the query graph and the relations to be joined. Therefore, we can successively join two relations until the number of joins is less than the number of relations in the folded join graph. In the worst case, we apply the folding steps until we are left with exactly two nodes and one edge, which by definition is acyclic. From this, it follows that the folding process eventually results in an acyclic join graph. □

4.4 Putting It All Together

With the transformation of cyclic queries into acyclic queries complete, we can now present our final ResultDB$_{\text{semi-join}}$ Algorithm 4.

Algorithm 4 `SELECT RESULTDB` on arbitrary join graph topologies.

```
1: function RESULTDB_SEMI-JOIN(G)
2:     if G.is_cyclic() then
3:         G = G.fold_join_graph(G)              ▷ transform into acyclic JG
4:     G = reduce_relations(G)                   ▷ use Yannakakis' algorithm
5:     for r ∈ G.relations do
6:         if r.is_fold() then                   ▷ r could be a join result
7:             base_rels = G.decompose(r)        ▷ split join into base relations
8:             G.deduplicate(base_rels)          ▷ remove potential dups
9:         else
10:            G.deduplicate(r)                  ▷ projection could introduce dups
11:    return G.relations                        ▷ result database
```

Given an arbitrary join graph, we first check if the join graph is cyclic (line 2). In that case, we use our folding algorithm (line 3) to construct an acyclic join graph. Afterward, we can reuse Yannakakis' algorithm to reduce the relations (line 4). At this point, a relation can also represent a join result, and the algorithm performs the semi-joins based on the modified join predicates. These join predicates can now be conjunctions of multiple joins predicates (cf. Figure 6). Finally, we have to break up the joins again and remove potential duplicates. For this, we iterate over all nodes in

the join graph (line 5) and check if this node is a fold (line 6). If so, we *decompose* this node again (line 7) by projecting each involved base relation and remove potential duplicates (line 8). In general, this operation can basically be seen as the inverse of the folding algorithm. We also remove duplicates due to the projection from base relations (line 10) and finally return the reduced relations (line 11).

Theorem 4.4. *The algorithm ResultDB$_{\text{semi-join}}$ (Algorithm 4) is correct, i.e., we obtain the correct and fully reduced relations.*

Proof. Let $Q \in \mathcal{Q}$ be a query over a set of relations $R = \{R_1, R_2, \ldots, R_n\}$ and $Q_{\text{ST}}(Q, R)$ be the single-table evaluation of Q and its inputs R (see Definition 2.1). Further, let $\textsc{Yannakakis}_{\text{reduce}}$ be the reduction phase of Yannakakis' algorithm depicted in Algorithm 2, let $\textsc{Decomp}\big(Q_{\text{ST}}(Q,R)\big) = \big\{\pi_{R_1}\big(Q_{\text{ST}}(Q,R)\big), \ldots, \pi_{R_n}\big(Q_{\text{ST}}(Q,R)\big)\big\}$ be the operation that decomposes a query result into its base relations, and let $\textsc{Decomp}_{\text{folds}}$ analogously split all folds into their base relations.

We have to show that applying Yannakakis' reduction phase to an arbitrary join graph yields the same result as performing the decomposition operation on the single-table result. In Lemma 4.3, we have already shown that an arbitrary query Q can be transformed into an acyclic query Q' with its corresponding set of relations R' without changing the query result, i.e., $Q_{\text{ST}}(Q,R) = Q_{\text{ST}}(Q',R')$. If Q is already acyclic, the folding process is a no-op. Let $JG_{Q'}$ be the corresponding join graph instantiated with the underlying relations R'. Since Q' is acyclic, we can apply Yannakakis' algorithm. Therefore, $\textsc{Yannakakis}_{\text{reduce}}(JG_{Q'})$ produces the fully reduced relations R'_{reduced}. Since the folds in R'_{reduced} are fully reduced as well, applying the $\textsc{Decomp}_{\text{folds}}$ operator to those folds yields the fully reduced base relations in turn. Therefore, we can conclude:

$$\textsc{Decomp}\big(Q_{\text{ST}}(Q,R)\big) = \textsc{Decomp}_{\text{folds}}\big(\textsc{Yannakakis}_{\text{reduce}}(JG_{Q'})\big)$$

□

5 Related Work

SQL Extensions. Throughout the years, various SQL extensions have been introduced. Regarding single keyword extensions, the Data Cube [19] and Skyline [9] operator are probably the most well-known. The Data Cube enables N-dimensional aggregate computation while the Skyline operator allows for filtering out "interesting" data points. Our work also introduces a new keyword, enabling the computation of a result subdatabase instead of a single-table result. To our knowledge, this is the first work to make this contribution.

Another way of extending SQL is to move away from the traditional relational data model and to introduce a query language tailored to semi-structured or unstructured data. SQL++ [32] is a semi-structured query language of AsterixDB [1] that represents a superset of the SQL and JSON data model. Extending SQL with JSON enables the arbitrary nesting and composition of data values. While this is a valid approach, our proposed extension explicitly stays in the relational word, keeping schema and relational information. Additionally, it is minimally invasive, requiring only the addition of a single keyword rather than a new query language.

Semi-joins & Yannakakis' Algorithm. To compute a result database, the fundamental idea is to use semi-joins to reduce all involved relations. Bernstein et al. [5] already introduce the idea to use semi-joins to solve relational queries. By "solve relational queries", they mean efficiently computing reduced relations using semi-joins and then computing the final result using these reduced relations. This algorithm is also well-known as Yannakakis' [40] algorithm. In their work, they show that queries with a tree-structured query graph can be solved using semi-joins. Conversely, queries with cyclic graphs generally require large *semi-join programs* or cannot be solved at all. Both works inspired our algorithmic solution.

Yang et al. [39] recently published another work utilizing Yannakakis' algorithm. The general idea is to optimize join performance by reducing the corresponding relations as much as possible before joining them. However, instead of using semi-joins to filter the relations, they utilize computationally more efficient Bloom filters. Their technique is called *predicate transfer* and can be seen as a generalization of Bloom joins. Unlike their approach, we cannot use Bloom filters without further ado due to potential false positives. While they remove false positives in the final join, we return the filtered relations directly. We could use such a probabilistic data structure only if the post-join is always executed on the client.

Bitmaps, Bitvectors, Bitmap Join Indexes, & Bloom Filters. Value bitmaps [11, 31], bitvector filters [15], bitmap join indexes [2], and their probabilistic counterparts like bloom filter [7, 24, 33] are useful optimization techniques for read-mostly scenarios. An early variant of this is the technique described by Graefe [18]. Especially bitvector filters and bloom filters can be seen as a variant of sideways information passing (see paragraph below). Since ResultDB essentially only filters the base relations of a query, any existing DBMS already supporting these techniques can directly apply them to ResultDB queries. A detailed experimental evaluation of the different techniques and trade-offs is beyond the scope (and space constraints) of this paper, but we believe that it will considerably improve the performance of ResultDB queries.

Sideways Information Passing & Factorization. The underlying idea of sideways information passing (SIP) is to optimize query processing by exchanging information between arbitrary parts in a query plan. This optimization is often aimed at reducing intermediate (join) results by *reducing* relations early on. Therefore, a semi-join reduction [5] is actually a special case of SIP, whose application is very broad. Shrinivas et al. [35] present how SIP in the context of materialization strategies can be used to improve the performance in the Vertica Analytical Database [23]. Neumann and Weikum [27] show how SIP can be used to speed up index scans at query runtime in RDF graphs. Another work [41] by Zhu et al. is able to produce robust query plans in star schemas by making use of bloom filters, a SIP data structure. While SIP applies a certain reduction in a specific scenario, we unconditionally reduce all relations that are part of a query. Furthermore, we explicitly compute this reduced state of a relation whereas SIP only uses it as a means to speed up query processing.

Factorized Databases [30] is another concept that tries to minimize intermediate join results. However, in contrast to the aforementioned SIP methods, factorization is essentially a compression technique for join results that tries to get rid of redundancies. With that, it also targets the problem of relational information redundancies (cf. Problem 1 in Section 1). In contrast to factorized databases,

we primarily try to avoid the redundancies in join results instead of hiding those redundancies using compression techniques.

Tree decompositions. As discussed in Section 4.3, tree decompositions [34] can be used to transform a cyclic graph into a tree structure by grouping nodes connected by edges into "bags", adhering to specific properties. While tree decompositions are useful, particularly in database theory [4, 8, 10, 17] for efficiently solving NP-hard problems like evaluating conjunctive queries with bounded treewidth, they have some practical issues. The main issue with tree decompositions is that they often result in trees where original nodes appear in multiple bags, increasing the join effort. To avoid this, we use a straightforward approach that joins (folds) relations until the graph is acyclic. This method provides better control over which relations are joined and allows for easier optimization with regard to the subsequent reduction phase.

Worst-case optimal joins. A key advantage of our algorithm presented in Section 4 is that we avoid computing potentially large intermediate results. Similarly, worst-case optimal joins (WCOJ) avoid producing large intermediate results by computing multi-way joins instead of binary joins [28]. However, WCOJ come with some major downsides. First, they are predominantly useful in contexts like large graphs with many self-joins [36]. Second, they are mainly efficient for cyclic queries [38]. Third, and most importantly, existing and well-known algorithms like the Leapfrog Triejoin [37] come with impractical requirements such as the existence of ordered index structures on their input. Freitag et al. [16] provide an approach to integrate WCOJ in relational databases without such hard requirements. While this work makes WCOJ more accessible outside the database theory community, we argue that Yannakakis' algorithm is better suited, as we can use it to efficiently compute multiple result sets while avoiding large intermediate join results.

Data Provenance. There are fascinating relationships of ResultDB to lineage and provenance. For instance, the seminal work by Cui et al. [14] introduced query rewrites to track the *derivation set* of an output tuple t, i.e., all tuples from the input database that contributed to computing t (see Definition 8.2 of Cui et al. [14]). Consider any SPJ query Q and reduce its output to a single tuple by applying filters, resulting in Q^t. Then, the derivation set of t *is equal to the ResultDB query* of Q^t by definition (as long as ResultDB returns all attributes from all referred input relations)! This is because we return only those tuples that somehow contribute to at least one of the tuples in the single-table result. In a way, ResultDB queries can be seen as *multi-tuple derivation set queries*.

Similarly, the work by Niu et al. [29] and Arab et al. [3] provides techniques for optimizing provenance queries. All these techniques should be revisited to leverage ResultDB style query processing. However, this goes beyond the scope of this paper.

6 Experiments

Through our experiments, we address the following four research questions (RQs):

RQ 1 What is the level of data redundancy in the result sets of a real-world benchmark, and what is the potential data redundancy in a theoretical scenario? (Section 6.1)

RQ 2 How do the RMs compare, and what is their overhead relative to the single-table approach? (Section 6.2)

RQ 3 What is the query execution time of our ResultDB$_{\text{semi-join}}$ compared to the single-table approach? (Section 6.3)

RQ 4 How does the end-to-end runtime of our ResultDB approach compare to that of the single-table approach, including data transfer time and post-join time? (Section 6.4)

Setup. All experiments were conducted on an AMD Ryzen Threadripper 1900X 8-Core processor with 32 GiB main memory. The underlying operating system is Arch Linux with kernel version 6.8.2.arch2-1 on an x86_64 architecture.

Systems. We integrated our ResultDB$_{\text{semi-join}}$ algorithm from Section 4 into mu*t*able [21], an open-source relational DBMS with a compiling query engine [22]. Currently, mu*t*able is an early-stage research project supporting core functionality like query execution, different data layouts, and plan enumerators [20]. However, mu*t*able is still missing some functionality like indexes, multi-threading, or advanced language features. Due to the lack of certain features, we decided to use PostgreSQL for evaluating the rewrite methods presented in Section 3. In particular, we use PostgreSQL 16.2 and measure the client-side runtime using the `\timing` command of the accompanying interactive terminal *psql*. Furthermore, we increase the *shared_buffers* size to 16 GiB and the *work_mem* size to 1 GiB based on empirical analysis, as these parameters significantly impact performance, and use default settings otherwise.

Datasets & Workloads. To evaluate our algorithms, we use the *Join Order Benchmark* (JOB) [25], which is based on the real-world IMDb dataset and exclusively contains SPJ queries with a variable number of joins. Some queries were slightly modified for use in mu*t*able due to missing keywords like `IN` or `BETWEEN`, without changing their semantics.

Query Types. We distinguish the following query types:

(1) *Single Table* (ST): refers to the SQL query that produces a potentially denormalized single-table join result and serves as a baseline (cf. Definition 2.1).
(2) *ResultDB without post-join information* (RDB): refers to the query that returns only the individual relations that are part of the `SELECT` clause of the original query, i.e., we exclusively project those attributes (cf. Definition 2.2).
(3) *ResultDB with post-join information* (RDB$_{\text{RP}}$): additionally projects the attributes necessary the post-join, effectively creating a relationship-preserving query (cf. Definition 2.3).

6.1 Result Set Sizes

In this section, we first have a closer look at the result set sizes of several JOB queries. Afterward, we will examine a star schema dataset to illustrate the potential extent of data redundancy. We investigate the result set sizes for the three different approaches ST, RDB, and RDB$_{\text{RP}}$. We compute the size of a result set by adding up the individual sizes of all attributes that are returned. The size of an attribute is calculated by multiplying the datatype size by the number of tuples for numeric attributes, or by summing the actual string length of each tuple for character attributes.

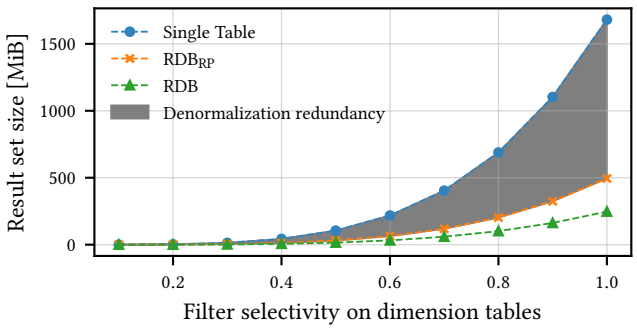


Figure 7: Theoretical star schema result set sizes.

JOB. Table 1 shows the result set sizes for a subset of the JOB queries in kilobytes (KiB), as well as the compression ratio defined as $\text{size(Single Table)}/\text{size(subdatabase)}$. Due to space constraints, we limit ourselves to those queries that we investigate in more detail later on in Section 6.4. Furthermore, the queries that we left out have mostly relatively small result set sizes, typically just a few kilobytes, and also exhibit small compression ratios. This is due to the fact that those JOB queries often return a very small number of rows, project a limited set of attributes, or both. In addition, there are very few redundancies in the result sets. Note that, the result set sizes of some queries like `4a` for RDB$_{RP}$ (19.07 KiB > 18.91 KiB) are even larger because they need to project additional attributes. The queries that stand out are `16b` with a very large result set size of roughly 130,376 KiB (≈ 127 MiB) and `11c` with a compression ratio of 1410. In general, most queries that return a subdatabase, regardless of whether they are relationship-preserving, tend to produce significantly smaller result sets. This, in turn, leads to high compression ratios due to substantial redundancy in the output.

Star Schema. To illustrate the extent to which data redundancy can occur, let us consider a typical star schema consisting of a fact table and several dimension tables. In the worst case, each tuple from a specific dimension table joins with all tuples from the other dimension tables, i.e., the fact table contains the Cartesian product of the dimension tables. This results in maximum redundancy. Assume we query the star schema by joining all relations, selecting all attributes, and scaling the result set size using filters with varying selectivity across all dimension tables. Figure 7 shows the result set sizes of this dataset and workload. Note that, in this case, RDB only projects the payload of the dimension tables and the fact table, meaning that no primary or foreign keys are returned. In comparison, both Single Table and RDB$_{RP}$ include this key information.

In general, the result set size for all three approaches increases quadratically with respect to the selectivity due to the characteristics of the fact table. However, the increase of the Single Table approach is much steeper since the data of all dimension tables gets repeated. The higher the selectivity value, the more tuples from the fact table participate in the join result and with that, more data from the dimension tables gets duplicated. As a result, the gap between Single Table and RDB$_{RP}$ widens as the selectivity value increases. This redundancy is depicted in gray. Clearly, the gap widens significantly the more dimension tables we have and the larger the tuples in the dimension tables are.

Regarding **RQ 1**, these results demonstrate that denormalization through joins, whether in JOB queries or in a theoretical star schema workload, can introduce a significant amount of data redundancy. This motivates computing and returning a subdatabase instead of a potentially denormalized single-table result.

6.2 Rewrite Methods

In this section, we investigate the performance of our rewrite methods introduced in Section 3 with respect to the end-to-end client-side runtime in PostgreSQL. For each JOB query template (1–33), we show the results for one specific instantiation (a–d), selected based on whether the query is supported by mu*t*able or chosen at random otherwise. We first compare the performance of the different rewrite methods against each other. Afterward, we examine the overhead or potential speed-up the rewrite methods introduce in comparison to the single-table execution. It is crucial to approach this comparison with caution, as the methods yield different results; it primarily aims to provide an intuition of the difference in query execution time. For this experiment, we use the RDB approach, i.e., we use the original query and do not add additional attributes for the post-join. To avoid data transfer overhead, we use `COUNT(*)` to aggregate the results. Since every rewrite method might consist of multiple queries, we summarize the measured runtime of the individual queries to obtain a single query execution time. For example, for RM 2 we first measure the time to create the MV and then the time to execute each `SELECT DISTINCT` query. We report the median query execution time of five runs.

Figure 8 shows the *query execution time* on a logarithmic scale for JOB. In general, it is evident that the rewrite methods exhibit significant differences in performance. No method is consistently better or worse than the others. The dynamic rewrites RM 1 and RM 3 exhibit very often a similar performance. While RM 1 is rather consistent in comparison to the other methods, RM 3 contains some noticeable outliers. For example, for `7a`, `21a`, `27a`, and `30c` the query execution time of RM 3 is up to an order of magnitude slower. However, RM 3 also has some positive applications, e.g., for `11c` or `20b`, where it outperforms the other RMs by far. Looking at the physical execution plan in PostgreSQL, we can see that RM 3 uses a different join order and essentially computes a semi-join between the outer relation and the subquery through the use of the `IN` keyword. In comparison to the other rewrites, RM 1 and RM 3 perform best if there is only a single relation referenced in the projections of the query, which is the case for `2a`, `3c`, `5c`, `11c`, `17a`, and `20b`. As soon as there are two or more relations, they fall behind. The materialization rewrites RM 2 and RM 4 exhibit very similar query execution times as well, with RM 4 being faster in almost all cases. This is most likely due to the fact that RM 4 simply requires less data to be materialized since it only materializes the join index instead of the complete join result (all required attributes).

Table 2 shows the *overhead* in percent for each query using the best performing rewrite method, comparing it to the single-table query execution time as baseline. A negative overhead represents an improvement. Except for a few queries like `15d` (43.2%), and

Table 1: JOB result set sizes in KiB (compression ratio).

Method	3c		4a		9c		11c		16b	
ST	166.16	(1.0)	18.91	(1.0)	497.56	(1.0)	267.85	(1.0)	130376.72	(1.0)
RDB_{RP}	101.66	(1.6)	19.07	(1.0)	57.07	(8.7)	0.19	(1409.7)	7884.80	(16.5)
RDB	101.66	(1.6)	13.15	(1.4)	35.29	(14.1)	0.19	(1409.7)	3821.99	(34.1)
Method	**18c**		**22c**		**25b**		**28c**		**33c**	
ST	686.85	(1.0)	783.46	(1.0)	0.25	(1.0)	310.66	(1.0)	9.32	(1.0)
RDB_{RP}	627.69	(1.1)	122.68	(6.4)	0.18	(1.4)	53.32	(5.8)	1.37	(6.8)
RDB	254.96	(2.7)	33.53	(23.4)	0.10	(2.5)	15.47	(20.1)	0.76	(12.3)

Table 2: Overhead of the best rewrite method compared to the single-table execution time for the IMDb dataset.

JOB Query	1b	2a	3c	4a	5c	6a	7a	8a	9c	10c	11c
Overhead	10.7%	0.8%	1.8%	12.9%	-2.0%	19.0%	0.6%	0.9%	23.4%	-0.6%	-84.1%
Best RM	RM 4	RM 3	RM 1	RM 4	RM 3	RM 4	RM 4	RM 4	RM 4	RM 4	RM 3
JOB Query	**12a**	**13b**	**14a**	**15d**	**16b**	**17a**	**18c**	**19a**	**20b**	**21a**	**22c**
Overhead	3.0%	1.8%	-1.2%	43.2%	48.6%	-53.5%	4.7%	2.6%	-90.5%	-1.7%	3.1%
Best RM	RM 4	RM 4	RM 4	RM 4	RM 4	RM 3	RM 2	RM 4	RM 3	RM 4	RM 4
JOB Query	**23a**	**24a**	**25b**	**26a**	**27a**	**28c**	**29a**	**30c**	**31a**	**32a**	**33c**
Overhead	-4.9%	0.9%	-0.0%	-75.8%	6.7%	1.2%	-0.0%	4.6%	-0.1%	9.5%	12.2%
Best RM	RM 4	RM 4	RM 4	RM 2	RM 4	RM 4	RM 4	RM 4	RM 4	RM 4	RM 4

Figure 8: Query execution time of the rewrite methods in PostgreSQL for the IMDb dataset.

16b (48.6%), at least one of the rewrite methods performs comparably to the single-table execution. For example, q2a and q31a have 0.8% and −0.1% overhead, respectively. However, there are also a few queries like 11c (−84.1%) and 20b (−90.5%) where our rewrite methods significantly outperform the single-table execution. In those cases, RM 3 is able to successfully apply a semi-join. This comparison underscores the small performance overhead of our approach, which is occasionally even faster.

Regarding **RQ 2**, we conclude that query execution times vary significantly across different rewrite methods depending on the type of query, with occasional outliers. However, most importantly, the best rewrite method often introduces only a marginal overhead over the single-table execution and sometimes even outperforms it. As a general rule, we recommend using RM 3 in case there is just a single output relation and RM 4 otherwise, as it is the best performing rewrite method in 75% of the cases.

6.3 ResultDB$_{\text{semi-join}}$ Algorithm

In this section, we evaluate the performance of our ResultDB$_{\text{semi-join}}$ algorithm described in Section 4 integrated into mu*t*able. Specifically, we compare the query execution time of our implementation with the single-table execution. To ensure a fair comparison despite the differing results of both approaches, we implemented a new logical/physical operator called *Decompose*. This operator is placed on top of the standard projection operator at the root of a plan. Instead of returning a single-table result set, it provides the ResultDB output by splitting the result into individual relations and removing duplicates. We briefly compare the ResultDB performance, in particular the post-join times, to the unchanged baseline in Section 6.4.

We implemented Algorithm 4 with one additional optimization. Once we fully reduced all relations that are part of the projections, we stop early and return the result, as there is no need to always

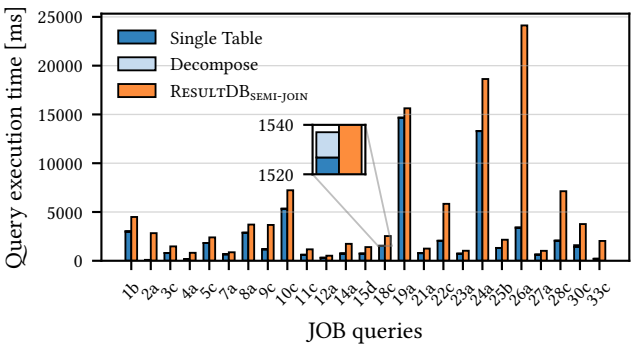


Figure 9: Query execution time of ResultDB$_{\text{semi-join}}$ in mutable for the IMDb dataset. The zoom-in highlights the negligible overhead of the decompose operation.

reduce all relations. In addition, we minimize data transfer by returning only the number of qualifying rows instead of the complete result. As for the rewrite methods, we use the RDB approach without adding attributes for the post-join. Furthermore, we inject the real cardinalities of filtered relations and join results to avoid negative effects on the query execution time due to poor cardinality estimation. Note that, mutable's query engine is currently limited to 16 GiB and does not support varying-length character data types. To run a meaningful amount of JOB queries, we manually limit the corresponding attributes in the IMDb schema to a fixed length, determined empirically. Nevertheless, some queries still exhaust this memory limit and therefore, we were not able to include all queries. We report the median query execution time of five runs.

Figure 9 compares the query execution time of Single Table plus Decompose as a stacked bar plot with ResultDB$_{\text{semi-join}}$ for several JOB queries. The key observation is that ResultDB is consistently slightly slower than the Single Table plus Decompose approach. Notably, due to small result set sizes, the Decompose step introduces hardly any overhead, making it nearly imperceptible in our plot. For instance, as depicted in the zoom-in, the decomposition for `18c` requires roughly just 10 ms, which is just a tiny fraction of the overall execution time. The same applies to all other queries in this experiment. Note that, preliminary experiments show that the Decompose step can be quite time-consuming for sufficiently large data sets. In general, for most queries the overhead introduced by our prototype implementation is still manageable. For example, for `q5c`, ResultDB$_{\text{semi-join}}$ has a runtime of 2396 ms, compared to 1821 ms for the single-table execution. However, there are also a few outliers were ResultDB$_{\text{semi-join}}$ performs considerably worse like for `26a` or `33c`.

Regarding **RQ 3**, we conclude that the current implementation of the ResultDB$_{\text{semi-join}}$ algorithm does not yet match the single-table execution times, even when factoring in the decomposition. However, despite our implementation lacking many optimization opportunities, such as improved folding or an optimized semi-join reduction order, these experiments yield promising results. Furthermore, the very small Decompose overhead essentially provides us with the possibility to compute a result subdatabase very efficiently.

6.4 Runtime with Data Transfer & Post-join

In this section, we compare the rewrite methods and our ResultDB$_{\text{semi-join}}$ algorithm to the single-table execution in terms of *end-to-end* runtime, including query execution, data transfer, and post-join time. For this experiment, we use RDB$_{\text{RP}}$ queries to be able to compute the post-join after transferring the data. Due to space constraints, we focus on a subset of the JOB queries, aiming to cover a diverse range of query types. The data transfer rate (DTR) – the time required to transfer a query result – plays a critical role in overall performance. For the following experiments, we assume a DTR of 100 Mbps, a speed commonly regarded as reliable for general use. While data centers and cloud-based networks often achieve DTRs in the tens or even hundreds of gigabits per second, these environments typically handle significantly larger datasets. As a result, our chosen DTR provides a meaningful basis for evaluating performance in more conventional settings. To measure the post-join times, we compute and materialize the reduced relations and use the respective system, PostgreSQL or mutable, to compute the final join result.

Rewrite Methods. Table 3 shows the end-to-end performance of the best rewrite method (RM) in PostgreSQL compared to the single-table (ST) approach. In general, for queries with larger result set sizes and high compression ratios (e.g. `9c`, `16b`, and `22c`), the transfer time is noticeably higher for the single-table result than for the result subdatabase. For example, transferring the single-table result of `16b` takes approximately 10, 186 ms, whereas transferring the result subdatabase requires only around 616 ms. Furthermore, the post-join times are almost negligible in the overall runtime and in comparison to the single-table execution. In particular, many JOB queries have post-join times of just a few milliseconds, which typically represents only a small fraction of the original single-table query execution time. However, some queries, such as `16b`, have high post-join times. In general, most queries exhibit similar performance across both approaches due to the small overhead introduced by the data transfer and the post-join. Note that, the query execution times are slightly higher than the ones reported in Figure 8 because we compute a relationship-preserving subdatabase. However, there are cases where the high data transfer time has a significant impact. For instance, the time for computing a result subdatabase for `16b` is almost twice as high as computing the single-table result, with an execution time of 10, 959 ms compared to 21, 560 ms. However, due to the reduced transfer time of approximately 616 ms – compared to nearly 10, 186 ms – and the small post-join time of 314 ms, the rewrite method almost performs comparably to the single-table execution in terms of overall execution time with 21, 145 ms versus 22, 490 ms.

ResultDB$_{\text{semi-join}}$. Because mutable's query execution is significantly faster compared to PostgreSQL, the post-join times are also considerably reduced. For most JOB queries in this experiment (excluding `16b` as it does not run in mutable), the post-join times consistently fall below 1 ms, with the highest one being 4.65 ms for `18c`. The transfer times are the same as the ones in Table 3. As a result, the overall query execution times are similar or higher (due to computing RDB$_{\text{RP}}$) to those in Figure 9, leading to the decision to omit the corresponding visualization due to space constraints. Regarding **RQ 4**, we conclude that in terms of end-to-end runtime,

Table 3: End-to-end performance of the best rewrite method (RM) compared to the Single Table (ST) execution on JOB.

JOB Query	3c		4a		9c		11c		16b	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	226.48	367.33	101.97	173.89	505.92	656.25	1098.98	195.59	10959.35	21559.84
Data Transfer [ms]	12.98	7.94	1.48	1.49	38.87	4.46	20.93	0.01	10185.68	616.00
Post-join [ms]	-	4.93	-	5.60	-	13.87	-	0.78	-	314.06
$\sum$ [ms]	239.46	380.20	103.44	180.98	544.79	674.58	1119.90	196.38	21145.03	22489.90
JOB Query	**18c**		**22c**		**25b**		**28c**		**33c**	
Approach	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**	**ST**	**RM**
Query Execution [ms]	2783.83	3217.37	2219.48	2356.31	250.47	360.65	2792.73	2886.34	90.56	123.34
Data Transfer [ms]	53.66	49.04	61.21	9.58	0.02	0.01	24.27	4.17	0.73	0.11
Post-join [ms]	-	35.67	-	6.16	-	2.45	-	6.83	-	11.22
$\sum$ [ms]	2837.49	3302.08	2280.69	2372.05	250.49	363.12	2817.00	2897.34	91.29	134.66

ResultDB approaches the single-table execution and that fast data transfer times can have a significant impact as for 16b.

7 Future Work

In the following, we suggest possible topics for further investigation that were beyond the scope of this initial work.

(1) **Query Optimization.** As discussed throughout this work, there is significant optimization potential yet to be explored, which our research group is actively investigating. We are working on several different algorithms that address the *Root Node Enumeration Problem* and the *Tree Folding Enumeration Problem*.
(2) **Data Transformations.** This initial paper focuses on the *data retrieval* aspect of SQL, i.e., SPJ queries. However, we also have ongoing research regarding *data transformation*. In this context, we plan to extend our definitions in Section 2 accordingly. We envision to integrate arbitrary data transformation alongside data retrieval, including grouping on different criteria on different relations **at the same time**. Consequently, our approach naturally supports true *grouping sets* semantics without shoehorning the different results into the same single output table.
(3) **Subqueries.** For non-correlated subqueries, we already explored the idea to naturally extend table-valued subqueries to *subdatabase-valued* subqueries. Instead of a single-table result, the outer query receives multiple reduced tables. For correlated subqueries, we plan to explore rewrite techniques similar to [26] and investigate how this affects query optimization.
(4) **Views.** In contrast to traditional views, a subdatabase view offers a view on a set of relations rather than a single relation. Similar to correlated subqueries, we want to investigate which optimization potential querying a subdatabase view exhibits.
(5) **Subdatabase Snapshot.** In this paper, returning a subdatabase is restricted to the relevant tuples of a subset of the tables. However, a result subdatabase could also include metadata, statistics, indexes, or the query execution plan for performing the post-join. For instance, the query execution plan could be sent in WebAssembly and executed within a sandbox by the client, eliminating the need for users to perform the post-join manually.
(6) **API Integration.** Current database APIs like JDBC drivers, expect a single-table result, i.e., cursor of tuples, to be returned. We propose a minimally invasive extension, enabling to return a set of cursors, with each cursor corresponding to a distinct result set. Further, we aim to explore the feasibility of a cursor that iterates over the join co-groups of multiple result sets, reducing the user's burden of performing the post-join on the client.

8 Conclusion

SQL comes with the very hard limitation that each query result is shoehorned into a single table. In this work, we initially discuss the fundamental problems, data redundancy and information loss, that stem from this limitation. To address these problems, we propose to extend the SQL SELECT clause by a single keyword: RESULTDB. This extension enables us to return a subdatabase – a subset of tables, each containing only the tuples that contribute to the overall query result – instead of a single, potentially denormalized, table. Furthermore, we introduce a formalization of our SQL extension showing that it is well-defined and has clear semantics.

We present two classes of approaches to support our new functionality. First, we propose four SQL-based rewrite methods allowing us to transform traditional SQL queries into semantically equivalent queries returning a result subdatabase. Second, we propose an efficient algorithm that can be integrated directly into a DBMS. We also show promising experimental results. Computing individual result sets can significantly reduce the result set size. Further, our RMs and algorithm introduce only minimal overhead and can sometimes even outperform the single-table execution.

Limitations. As already pointed out in Section 7, this work opens the book for a lot of exciting follow-up works that should investigate some of its limitations. Our experiments show that our native algorithm is slower than the single-table execution. However, we also showed that we can use our decompose operator, a simple extension to any traditional query execution plan, to efficiently compute a result subdatabase. Although ResultDB is slower, the comparison is not entirely fair. Single-table queries benefit from 50 years of extensive research, while the application of Yannakakis' algorithm in real systems experienced little optimization effort so far. However, we are confident that future work can address many of these challenges. In addition, this work is currently limited to SPJ queries. However, we already have ongoing research that tackles data transformation like aggregation and arithmetic expressions.

Acknowledgements. We thank Luca Gretscher for his support in implementing the presented algorithm in mutable as well as Karl Bringmann and Simon Rink for their suggestions and feedback. We would also like to thank the anonymous reviewers for their constructive feedback.

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[2] Kamel Aouiche, Jérôme Darmont, Omar Boussaid, and Fadila Bentayeb. 2005. Automatic Selection of Bitmap Join Indexes in Data Warehouses. In *Data Warehousing and Knowledge Discovery, 7th International Conference, DaWaK 2005, Copenhagen, Denmark, August 22-26, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3589)*, A Min Tjoa and Juan Trujillo (Eds.). Springer, 64–73. https://doi.org/10.1007/11546849_7

[3] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A Generic Provenance Middleware for Queries, Updates, and Transactions. In *6th Workshop on the Theory and Practice of Provenance, TaPP'14, Cologne, Germany, June 12-13, 2014*, Adriane Chapman, Bertram Ludäscher, and Andreas Schreiber (Eds.). USENIX Association. https://www.usenix.org/conference/tapp2014/agenda/presentation/arab

[4] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2022. *Database Theory*. Open source at https://github.com/pdm-book/community.

[5] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40. https://doi.org/10.1145/322234.322238

[6] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. 1981. Query Processing in a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.* 6, 4 (1981), 602–625. https://doi.org/10.1145/319628.319650

[7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[8] Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* 25, 6 (1996), 1305–1317. https://doi.org/10.1137/S0097539793251219

[9] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 421–430. https://doi.org/10.1109/ICDE.2001.914855

[10] Karl Bringmann, Nofar Carmeli, and Stefan Mengel. 2022. Tight Fine-Grained Bounds for Direct Access on Join Queries. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 427–436. https://doi.org/10.1145/3517804.3526234

[11] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, Laura M. Haas and Ashutosh Tiwary (Eds.). ACM Press, 355–366. https://doi.org/10.1145/276304.276336

[12] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[13] E. F. Codd. 1971. Further Normalization of the Data Base Relational Model. *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[14] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227. https://doi.org/10.1145/357775.357777

[15] Bailu Ding, Surajit Chaudhuri, and Vivek R. Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2011–2026. https://doi.org/10.1145/3318464.3389769

[16] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[17] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM* 56, 6 (2009), 30:1–30:32. https://doi.org/10.1145/1568318.1568320

[18] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611

[19] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. 1996. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, Stanley Y. W. Su (Ed.). IEEE Computer Society, 152–159. https://doi.org/10.1109/ICDE.1996.492099

[20] Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1 (2023), 73:1–73:26. https://doi.org/10.1145/3588927

[21] Immanuel Haffner and Jens Dittrich. 2023. mutable: A Modern DBMS for Research and Fast Prototyping. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p41-haffner.pdf

[22] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 1–13. https://doi.org/10.48786/EDBT.2023.01

[23] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[24] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *Proc. VLDB Endow.* 12, 5 (2019), 502–515. https://doi.org/10.14778/3303753.3303757

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[26] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings (LNI, Vol. P-241)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). GI, 383–402. https://dl.gi.de/handle/20.500.12116/2418

[27] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 627–640. https://doi.org/10.1145/1559845.1559911

[28] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, 37–48. https://doi.org/10.1145/2213556.2213565

[29] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, and Venkatesh Radhakrishnan. 2017. Provenance-Aware Query Optimization. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 473–484. https://doi.org/10.1109/ICDE.2017.104

[30] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[31] Patrick E. O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Joan Peckham (Ed.). ACM Press, 38–49. https://doi.org/10.1145/253260.253268

[32] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 http://arxiv.org/abs/1405.3631

[33] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14 (2009). https://doi.org/10.1145/1498698.1594230

[34] Neil Robertson and Paul D. Seymour. 1986. Graph Minors. II. Algorithmic Aspects of Tree-Width. *J. Algorithms* 7, 3 (1986), 309–322. https://doi.org/10.1016/0196-6774(86)90023-4

[35] Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the Vertica analytic database: Lessons learned. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1196–1207. https://doi.org/10.1109/ICDE.2013.6544909

[36] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (2022), 877–890. https://doi.org/10.14778/3574245.3574270

[37] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR* abs/1210.0481 (2012). arXiv:1210.0481 http://arxiv.org/abs/1210.0481

[38] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2 (2023), 150:1–150:23. https://doi.org/10.1145/3589295

[39] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf

[40] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[41] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *Proc. VLDB Endow.* 10, 8 (2017), 889–900. https://doi.org/10.14778/3090163.3090167