PsiDB: A Framework for Batched Query Processing and Optimization

Mehrad Eslami
Dept. of Computer Science
University of South Florida
Tampa, Florida, USA
mehrad@mail.usf.edu

Yicheng Tu
Dept. of Computer Science
University of South Florida
Tampa, Florida, USA
ytu@mail.usf.edu

Hadi Charkhgard
Dept. of Industrial Engineering
University of South Florida
Tampa, Florida, USA
hcharkhgard@usf.edu

Zichen Xu
Dept. of Computer Science
Nanchang University
Nanchang, Jiangxi, China
xuz@ncu.edu.cn

Abstract—Techniques based on sharing data and computation among queries have been an active research topic in database systems. While work in this area developed algorithms and systems that are shown to be effective, there is a lack of logical foundation for query processing and optimization. In this paper, we present PsiDB, a system model for processing a large number of database queries in a batch. The key idea is to generate a single query expression that returns a global relation containing all the data needed for individual queries. For that, we propose the use of a type of relational operators called $\psi$-operators in combining the individual queries into the global expression. We tackle the algebraic optimization problem in PsiDB by developing equivalence rules to transform concurrent queries with the purpose of revealing query optimization opportunities. Centering around the $\psi$-operator, our rules not only covered many optimization techniques adopted in existing batch processing systems, but also revealed new optimization opportunities. Experiments conducted on an early prototype of PsiDB show a performance improvement of up to 36X over a mainstream commercial DBMS.

Index Terms—Batch processing, Query processing, Query optimization, Equivalence rules

I. INTRODUCTION

Traditional database management systems (DBMSs) follow a one-query-at-a-time query processing model, in which queries in a workload are optimized and executed largely in an independent manner. Today’s database systems often need to serve a multitude of user queries in a short period of time. For example, a typical OLAP system in Alibaba, a leading Chinese online retailer, needs to process a workload at the 10,000 query per second (qps) level. Under such high demand, the one-query-at-a-time model falls short in meeting performance requirements due to resource contentions. On the other hand, the database community observed that queries in a workload may have a significant overlap in the data access paths and intermediate query results. Plausible efforts to harness computation and data access sharing among concurrent queries have been reported [1]–[7]. Although such work (reviewed in Section VI) comes with different flavors, in this paper, we call them concurrent database management systems (CDBMS) (see Section VI for a short review).

In CDBMSs, the main task in query optimization is essentially to find a global execution plan that combines the necessary computations to derive the results of multiple concurrent queries with the smallest resource consumption. While CDBMS systems are shown to significantly outperform traditional RDBMSs in both throughput and query response time [5]–[8], there is a lack of systematic consideration of query optimization. In particular, existing work took the approach of developing rules or heuristic algorithms for physical plan generation and selection. The representation and transformation of queries at the logical level is missing. As a result, existing CDBMSs fall short in the following two aspects.

First, they still leave much room for improving system performance due to the lack of a mechanism to harness a comprehensive set of optimization opportunities. This is because algebraic optimization, as an essential component in any query optimizer, provides opportunities that cannot be captured at the physical optimization stage. Second, they only work for fixed query workloads. The global query plan is often “handcrafted” for a workload with a static set of query templates [6], [7]. When new types of queries arrive, the global plan has to be recompiled instead of constructed on-the-fly. This is largely caused by the high optimization cost in working with complex query operator networks formed by multiple queries. In this paper, we address the above issues by introducing a new approach named PsiDB.

Our Contributions: Algebraic representation of queries is often seen as the first step towards query optimization. Following that, equivalence rules are used to transform the initial algebraic expression into others that also return the same results. Such rules are essential in deriving a rich set of query plans with different performance profiles for the choice of further steps in query optimization. In this paper, we argue that the rules adopted in traditional DBMSs are insufficient in capturing optimization opportunities in concurrent query processing. The main problem is that an algebraic expression should return multiple relations as output by following the design of existing CDBMSs, yet a vector of relations is not supported in current relational database theory. In particular, we make the following contributions.

First, we present the PsiDB system model for query processing in CDBMSs. The key idea is to generate a single query expression that returns a global relation containing all the data needed for individual queries. We also study the features of a group of relational operators called $\psi$-operators for gluing the
individual queries into the global expression;

Second, we develop equivalence rules to transform algebraic expressions within the PsiDB framework. Centering around the $\psi$-operator and a new concept of weak equivalence, our rules cover many optimization techniques adopted in existing CDBMSs;

Third, we show that PsiDB also reveals new opportunities that result in significant increase of workload processing throughput. Specifically, PsiDB favors multi-table joins as the main building block of the global algebraic expression. The efficiency of such expressions, although a surprise to many, can be easily justified by recent research findings.

We evaluate the PsiDB approach with extensive experiments running on an early prototype of PsiDB with realistic query workloads and obtained exciting results. The system is tested under different workloads with various batches of queries with distinct characteristics. The PsiDB design gives us a remarkable advantage in saving disk IO and sequential scan, as it results in the performance improvement shown in the experimental results evaluation section. For example, we observed a speedup of up to 36X for Select queries and 20x for Group-by/Aggregate queries over the latest version of SQL Server in processing various TPCH-style workloads. This is significantly higher than the 1.2-10X reported speedup of SharedDB over the less performant MySQL. Additionally, to determine the lower bound of the system we observed a speedup of 1.4x in our worst case scenario, over one-query-at-a-time by SQL Server

Paper Organization: In Section II, we introduce the PsiDB framework for batched query processing; In Section III, we present a series of equivalence rules geared towards concurrent query processing; In Section IV, we present key challenges in the design of a PsiDB query optimizer and present empirical evaluation results in Section V; In Section VI, we review related work in this field; We conclude this paper and comment on future directions in Section VII.

II. The PsiDB Framework

This section introduces a system model for concurrent query processing including all the definitions and terminology that are used throughout the paper. Note that our model differs from those of recent CDBMSs, especially DataPath [6] and SharedDB [7], in that we focus on algebraic optimization rather than physical optimization. There are also other differences that we will highlight in text marked Comparison.

A. Multi-Query-One-Execution Model

A traditional RDBMS processes one query at a time and returns a relation as the query result. The query processor of a DBMS needs to transform a query written in a declarative language such as SQL to a relational algebraic representation for further consideration in the query optimization stage. The same relational algebraic representation is needed for CDBMSs, the major difference is that there are more than one query to be executed at a time. Here we consider the situation that the CDBMS system organizes queries into batches and each batch consists of all queries that arrive during a period of time, same as in SharedDB [7]. Let us first define notations to set up the theoretical framework.

Definition 1. A table schema $R$ is a set of attributes, each of which defines a domain of values. A relation (table) $r$ is a set of tuples, each tuple $t$ (denoted as $t \in r$) has value following the schema $R$.

Definition 2. A query $q$, is a relational algebraic expression over a set of relations. The set $q = \{q_1, q_2, ..., q_n\}$ indicates the list of registered queries in the system to be executed concurrently.

Definition 3. An attribute is a column of a relation and is denoted as $A_x$. The resulting relation of query $q$, contains a set of attributes $L_q = \{A_1, A_2, ..., A_n\}$. This set $L_q$ includes all the attributes participating in the query projection and selection. We define a global attribute set containing all attributes in the resulting relations of all the $n$ queries as $L_q = L_1 \cup L_2 \cup ... \cup L_n$.

A database contains a set of relations $T = \{T_1, T_2, ..., T_m\}$, each of which contains different attributes. Each query will take one or a combination of such relations as inputs. As in many relational algebraic studies, we assume the relations in $T$ are designed following at least the 1st Normal Form. Due to page limits, we set the scope to queries with a selection-projection-join (SPJ) pattern. Actually, we also developed rules for group by and aggregates, and more details of such can be found in our technical report [9].

Example 1. We present a small database along with a few queries to be used for examples throughout the paper. The database consists of three tables: Part, PartSupp and Supplier. Hereafter, we will refer to them as P, PS and S, respectively.

There exists the following foreign keys: $PS.pkey \rightarrow P.pkey$ and $PS.skey \rightarrow S.skey$, and $(pkey, skey)$ serves as the primary key of $PS$. The query workload consists of five queries shown in Table II.

In a CDBMS, an expression should return a vector of relations (vs. one single relation in RDMS) as output. However, the existence of multiple output relations makes mathematical reasoning difficult, as no existing relational algebraic work supports vector of relations. Our approach is to introduce a global relation $\mathcal{P}$, which is a single relation that contains all the data needed to derive the final results for all queries in the workload. Therefore, our goal is to transform the individual queries (i.e., expressions) $q_1, q_2, ..., q_n$ into a single algebraic...
expression $Q$ whose resulting relation is $\mathcal{P}$. Each tuple in $\mathcal{P}$ will then be distributed to filter functions that belong to individual queries to generate the final result for the latter. Fig. 1 illustrates the PsiDB system design, and we elaborate on key components of the model as follows.

B. Combining Multiple Relations

With the concept of a global relation $\mathcal{P}$, it is necessary to develop an approach to combine results of individual relational expressions into $\mathcal{P}$. Specifically, we need to define a binary relational operator for such purposes. Intuitively, such an operator: (i) should generate a global output relation $\mathcal{P}$ that contains all the data of the operands; (ii) we can apply a filtering function to recover the exact individual operands; (iii) Furthermore, to apply the operation to a series of operands, the order of such sequence should not change the recoverability of all operands from $\mathcal{P}$. It turns out more than one relational operators satisfy such conditions. Let us first formalize the definition of such operators.

**Definition 4.** Given a binary relational operator $\odot$ and for $\mathcal{P} = T_1 \odot T_2$, the operator is a $\psi$-family operator (or simply $\psi$-operator) if all of the following conditions are satisfied: (i) **Operand Recoverability:** There exist relational algebraic functions $f_1$ and $f_2$ to recover the two input tables $T_1$ and $T_2$ exactly from $\mathcal{P}$, i.e., we must have $T_1 = f_1(\mathcal{P})$ and $T_2 = f_2(\mathcal{P})$; (ii) **Weak Commutativity:** Given $\mathcal{P}' = T_2 \odot T_1$, we can use the same functions $f_1$ and $f_2$ to recover $T_1$ and $T_2$ from $\mathcal{P}'$, i.e., $T_1 = f_1(\mathcal{P}')$ and $T_2 = f_2(\mathcal{P}')$; (iii) **Weak-Associativity:** The same functions $f_1$, $f_2$, and $f_3$ can be used to exactly recover $T_1$, $T_2$, $T_3$ from $\mathcal{P} = T_1 \odot (T_2 \odot T_3)$ and also from $\mathcal{P}' = (T_1 \odot T_2) \odot T_3$.

Note that the common concepts of commutativity (i.e., $T_1 \odot T_2 = T_2 \odot T_1$) and associativity (i.e., $T_1 \odot (T_2 \odot T_3) = (T_1 \odot T_2) \odot T_3$) are strong forms of conditions (ii) and (iii), respectively. In other words, when we reorder the sequence of operands of the $\psi$-operator, an unique global relation is not required. Instead, we only require that the same filter functions can be used to generate the original relations. Now we study individual members of the $\psi$-family.

**Theorem 1.** A Cartesian (or cross) product between two non-empty relations $T_1$ and $T_2$, defined as

$$T_1 \times T_2 = \{ t_1 \times t_2 : \forall t_1 \in T_1 \land \forall t_2 \in T_2 \}$$

is a $\psi$-operator.

**Proof.** We first check **Operand Recoverability:** suppose we have $T_1 \times T_2 = \mathcal{P}$ and $L_{T_1}, L_{T_2}$ are the set of attributes for relations $T_1$ and $T_2$, respectively. The exact input relations of the product operator can be easily recovered from $\mathcal{P}$ as

$$T_1 = \Pi_{L_{T_1}}(\mathcal{P}) \quad , \quad T_2 = \Pi_{L_{T_2}}(\mathcal{P}) \quad (1)$$

**Commutativity and Associativity:** It is easy to see that cross product is (strong) commutative and (strong) associative. Detailed proof can be found in [10, page 79].

A more interesting $\psi$-operator involves the concept of outer join. Let us first start with some definitions.

**Definition 5.** Selection condition $\theta$ on a relation $r$ compares two atomic values of $\theta = A_i \phi A_j$ or $\theta = A \phi e$ where $\phi = \langle, =, \neq, >, \land \rangle$ and $e$ is a valid value for the attribute $A_i$.

**Definition 6.** A selection operation over a relation $r$ is to apply a set of conditions as follows: $\sigma_\theta(r) = \{ t \in r \land \theta(t) = \text{TRUE} \}$.

**Definition 7.** The (inner) join between two relations by a common condition $c$ is

$$T_1 \bowtie T_2 = \sigma_{c}(T_1 \times T_2)$$

**Definition 8.** A left outer join between two relations is denoted as $T_1 \bowtie \left. \right. T_2$ where $c$ is the join condition. Let $L_{T_1}$ be the attribute list of table $T_1$ and let $\omega_2 = \{ \text{null}, \ldots, \text{null} \}$ be the singleton on all attributes of $T_2$, the left outer join can be described as follows:

$$T_1 \bowtie T_2 = (T_1 \bowtie c T_2) \cup \left( (T_1 - \Pi_{L_{T_2}}(T_1 \bowtie \left. \right. T_2)) \times \omega_2 \right)$$
Similarly, we have the right outer join defined as:

\[ T_1 \bowtie_Z T_2 = (T_1 \bowtie_Z T_2) \cup ((T_2 - \Pi_{L_{T_2}}(T_1 \bowtie T_2)) \times \omega_1) \]

**Definition 9.** A full outerjoin is a union between both left and right outerjoin (definition 8) on the same relations:

\[ T_1 \bowtie_Z T_2 = (T_1 \bowtie_Z T_2) \cup (T_1 \bowtie_L T_2) \]

**Theorem 2.** The full outerjoin \((\bowtie_Z)\) is a \(\psi\)-operator.

**Proof.** We consider \(P = T_1 \bowtie_Z T_2\) with any general join conditions \(c\) (Fig. 2a).

**Operand Recoverability:** The resulting table of a full outerjoin essentially consists of three parts: the inner join results, the tuples of \(T_1\) that do not join, and those of \(T_2\) that do not join. Since the schema of the resulting relation consists of all attributes in both tables, we only need to project out the corresponding set of attributes for \(T_1\) (or \(T_2\)) to recover \(T_1\) (or \(T_2\)). Formally, we have

\[ T_1 = \sigma_{A_1 \neq \text{null}}(L_{T_1}, P), \quad T_2 = \sigma_{A_1 \neq \text{null}}(L_{T_2}, P) \]

(2)

where \(A_1\) is an attribute in \(L_{T_1}\) or \(L_{T_2}\). The selection step after projection is to remove the tuples that contain null values in all attributes (e.g., unshaded regions in Fig. 2a).

**Commutativity:** By its definition, it is easy to show that full outerjoin is (strong) commutative: among the three sets of tuples in the resulting table, the inner join is known to be commutative [10]; that also leaves the tuples from \(T_1\) and \(T_2\) that do not appear in the inner join unchanged when we switch the operands.

**Associativity:** According to [10, page 96], the full outerjoin is strong associative with certain assumptions made with respect to the join conditions. We prove that it is weak associative under any general join conditions (see Appendix A for details).

**Comparison:** The way to combine queries together is a vital design decision of a CDBMS. Systems such as SharedDB [7] meet this challenge by combining intermediate query results with the union \((\cup)\) operator. However, this excludes operands that are not union compatible and thus is not as general a solution as \(\psi\). Various multi-query-optimization (MQO) approaches [11] focus on combining common subexpressions among queries while PsiDB works on entire query expressions. Certainly, there are practical issues with the use of \(\psi\), one major concern being the size of its output table. We will develop various techniques to address such a challenge.

1) **Size of Resulting Relation:** With the \(\psi\)-operator as a key concept in our framework, the size of its resulting table becomes an important query optimization consideration. In that sense, the cross product is in obvious disadvantage over outer join. Another disadvantage of the cross product is that neither operand can be empty (Theorem 1). Therefore, unless specified otherwise, our discussions in subsequent sections will be focused on the outer join.

An interesting observation is that we can control the size of outer join results via different join conditions. Specifically, the selectivity of the join condition determines how many times a tuple in \(T_1\) or \(T_2\) will appear in the final result (middle region shown in Fig. 2a). To minimize such redundancy, we could apply a dummy condition to disallow any joined tuples.

**Definition 10.** Given two tables \(T_1\) and \(T_2\), an anti-join condition \(\theta\) ensures \(T_1 \bowtie \theta T_2 = \emptyset\), and \(T_1 \bowtie \theta T_2 = T_1 \times \omega_2\).

In practice, the anti-join condition can be implemented as a system artifact or a real join condition that never returns true in any tuple (e.g., comparing an attribute with infinity).

**Definition 11.** For any two input tables, an outerjoin with join conditions that never returns a tuple in the inner join section is called an exclusive outerjoin; otherwise it is called an inclusive outerjoin.

In the remainder of this paper, unless stated otherwise, when we mention an outerjoin without a join condition, we mean the reasoning is true under arbitrary conditions.

Another factor to consider is the size of the \(\omega\) values – in modern DBMSs the cost of a NULL value is 1 bit [12], [13]. Thus, tuples appear in the inner join section can potentially decrease the number of NULL values. In an extreme case, all tuples in \(T_1\) and \(T_2\) participate in a 1-to-1 join, and the size of the outerjoin is minimized (the same as the total size of both input tables). In practice, for equi-joins or more strictly, natural joins, we could store only one copy of the common attributes to further save storage space (Fig. 2b).

In considering the use of \(\psi\) to combine different query expressions, we further show that the size of the final table can be effectively controlled via the adaptation of various equivalence rules (Section III) in query optimization.

**C. Filtering the Global Query**

As shown earlier, the global query \(Q\) can be written as

\[ q_1 \psi q_2 \psi \ldots \psi q_m \equiv Q \]

For ease of presentation, we name our framework PsiDB to highlight the central role the \(\psi\) operator plays. Upon generating the global relation \(P\), PsiDB uses a query result distributor to send the tuples of \(P\) to query filters associated with each original query \(q_i\). The latter will check each tuple with
predefined recovering functions and make on-the-fly decisions on what tuples/columns belong to its corresponding query result \( p_i \) (more details in Section IV-A2).

**Comparison:** In SharedDB and DataPath, each tuple in the global relation is associated with a bitmap describing which queries the tuple belongs to. Then the tuple is only multicast to corresponding queries. In PsiDB, we do not tag the tuples with its destination queries, therefore the tuples are broadcasted to all queries. We will provide an analysis of the performance of these two schemes in Section IV-A2.

**III. Equivalence Rules in PsiDB**

An equivalence rule essentially transforms a relational algebraic expression to another logically equivalent expression. They enable different query execution plans to be generated and evaluated thus form the foundation for query optimization. In this section, we extend work in traditional relational algebra to consider equivalence rules related to \( \psi \) operators and concurrent queries. Specifically, we study all the popular equivalence rules that are widely adopted in traditional DBMSs [14]. Based on such rules, we develop new rules that work particularly for batched queries. Furthermore, we discuss the impact of each rule on query optimization in text segments marked "Significance." We show that our rules can cover the main optimization techniques used in SharedDB and DataPath, and also provide extra optimization paths that are not found in such systems. Before beginning with the rules, we need to define another key concept of this paper.

**Definition 12.** Suppose a relational algebraic expression \( E_1 \) contains a series of \( \psi \) operators over multiple operands, i.e., \( E_1 = e_1 \psi e_2 \psi \ldots e_m \), the expression is said to be weakly equivalent to another expression \( E_2 \), denoted as \( E_1 \equiv E_2 \), if all the operands \( e_1, e_2, \ldots, e_m \) of \( E_1 \) can be exactly recovered from the resulting relation of \( E_2 \).

Via weak equivalence, an expression can be transformed into one that, although does not derive the same exact resulting relation, carries all the data and a means to recover the operands (of the original expression). Note that such transformations are not allowed in traditional DBMSs. In practice, we prefer a common function be used to recover the operands from both \( E_1 \) and \( E_2 \) in our equivalence rules. The following definitions are needed for describing the rules.

**Definition 13.** Suppose a query \( q_i \) applies a selection with a set of conditions in conjunctive form (denoted as \( \Theta_i \)) over a relation. Such selection conditions in all \( n \) concurrent queries over a relation is then denoted as \( \Theta_r = \Theta_1 \cup \Theta_2 \cup \ldots \cup \Theta_n \).

**Example 2.** The conditions sets for query workload as shown in Table II are as follows.

\[
\begin{align*}
\Theta_1 & = \{(\text{retprice}<40)\} \\
\Theta_2 & = \{(\text{supprice}<\text{retprice}), (\text{supprice}<40), (C(S,P), S.skey)\} \\
\Theta_3 & = \{(\text{supprice}<\text{retprice}), (\text{avlqty}>0), (C(S,P), S.skey)\} \\
\Theta_4 & = \{(\text{supprice}<\text{retprice}), (\text{state}=\text{FL}), (C(S,P), S.skey)\} \\
\Theta_5 & = \{(\text{state}=\text{FL}), (C(P,S), S.skey)\}
\end{align*}
\]

where \( C(A,B) \) represents equality join conditions set on foreign keys between tables \( A \) and \( B \) for easier presentation, e.g., \( C(S,P) = \{S.skey = PS.skey\} \). The total set of conditions for all 5 queries is

\[
\Theta_r = \{(\text{retprice}<40), (\text{supprice}<\text{retprice}), (\text{supprice}<40), (\text{avlqty}>0), (\text{state}=\text{FL}), (C(S,P), S.skey)\}
\]

**Definition 14.** The common selection conditions among all \( n \) concurrent queries over a relation is denoted as \( \Theta_r = \bigcap_{i=1}^{n} \Theta_i \). We also define \( \Theta'_j \) as the set of selection conditions for a query \( q_j \) (over a relation) that is not found in \( \Theta_j \), i.e., \( \Theta'_j = \Theta_j - \Theta_j \).

**Definition 15.** We can write an expression \( \theta_1 \) with all members of \( \Theta_1 \) in a conjunctive form, i.e., \( \theta_1 = \bigwedge_{i=1}^{n} (\Theta_1) \). In the same way, we define an expression \( \theta'_j \) with all \( \theta'_j \) expressions in a disjunctive form, i.e., we have \( \theta_1 = \bigvee_{i=1}^{n} \theta'_i \).

**A. Rule 1: Sequence of Selections**

In traditional DBMSs, we have

\[
\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} (r) \equiv \sigma_{\theta_1} (\ldots (\sigma_{\theta_n} (r)))
\]

and this essentially means the conditions can be applied to the relation one by one in any order. Such rules reveal the high level of flexibility in reordering different selection conditions over the same relation of the same (only) query. This is beneficial to query optimization as it provides a rich set of computational paths (i.e., execution plans) with different costs for the same set of selections. For example, a common practice is to process conditions with higher selectivity first to reduce the size of intermediate results. In this paper, we are interested in the flexibility of doing the same while processing batched queries. First, we observe that reordering of selection conditions cannot be done freely across different queries. In other words, we have

\[
\sigma_{\theta_0} (r) \psi \sigma_{\theta_1} (r) \psi \ldots \psi \sigma_{\theta_n} (r) \neq \sigma_{\theta_0} (\sigma_{\theta_1} (\ldots (\sigma_{\theta_n} (r))))
\]

and the above is true for any ordering of \( \sigma_{\theta_0} \) on the RHS.

Fortunately, the following theorem shows that there is room for rearranging the selection conditions among concurrent queries.

**Theorem 3 (Sequence of Selections).** For \( n \) concurrent queries, each of which applies a selection to the same table \( r \), we have

\[
\sigma_{\theta_1} (r) \psi \sigma_{\theta_2} (r) \psi \ldots \psi \sigma_{\theta_n} (r) \equiv \sigma_{\theta_1 \wedge \theta_2 \ldots \wedge \theta_n} (r)
\]

*Proof.* For weak-equivalence we need to show that each operand on the LHS must be recoverable from the resulting relation of both sides. Let us recall that table \( r \) follows schema \( R \). For the LHS, its resulting table \( P \) is a wide relation with repetitive occurrences of \( R \) in its schema, one for each operand.
Let us denote such occurrences as \( R \), we can recover each individual operand via

\[
\sigma_{\theta_i}(r) = f_i(P) = \Pi_R \sigma_{\theta_i}(P), \quad \forall i
\]

Note that the above is true for both cross product and full outerjoin under any join conditions.

For the RHS, we denote its resulting table as \( P' \), we show that we can use the function \( f'(P') = \sigma_{\theta_i}(P') \) to recover all operands of the LHS as follows.

\[
\sigma_{\theta_i}[\sigma_{\theta_1 \lor \theta_2 \lor \cdots \lor \theta_n}(r)] = \sigma_{\theta_i}[\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) \cup \cdots \cup \sigma_{\theta_n}(r)] = \sigma_{\theta_i}(r), \quad \forall i
\]

The recoverability of all operands of LHS concludes the proof.

From the above definitions, we get that the selection conditions over a relation for query \( i \) can be written as

\[
\theta_i = \bigwedge_{all}(\Theta_i) = (\bigwedge_{all}(\Theta_i')) \land (\bigwedge_{all}(\Theta_i)) = \theta'_i \land \theta_i
\]

(6)

Therefore, the RHS of Eq. (5) becomes \( \sigma_{\theta_1 \lor \theta_2}(r) \), and we can follow the traditional equivalence rules to make the following claims.

**Corollary 1.** The selection conditions within \( \Theta_i \) can be applied sequentially on the relation \( r \), according to Eq. (5).

**Corollary 2.** The selection condition sets \( \theta_D \) and \( \theta_1 \) can be applied sequentially, and in any order.

\[
\sigma_{\theta_1 \lor \theta_2}(r) = \sigma_{\theta_2}(\sigma_{\theta_1}(r)) = \sigma_{\theta_1}(\sigma_{\theta_2}(r))
\]

**Example 3.** In our example database, if we take a subset of the queries \( q = \{q_2, q_3, q_4\} \) as our workload, we get

\[
\theta_1 = \{(\supprice < \retprice) \land C_{(PS)} \land C_{(S)}\}
\]

and \( q'_2 = \{(\supprice > 40)\} \), \( q'_3 = \{(\avqty > 0)\} \) and \( q'_4 = \{(\state = FL)\} \) therefore the non-common conditions are

\[
\theta_D = \{(\supprice > 40) \lor (\avqty > 0) \lor (\state = FL)\}
\]

**Significance:** Rule 1 shows an important way to reduce the size of the results of the \( \psi \) operator. Consider the full outerjoin as \( \psi \), the total size for the LHS of Eq. (5) is \( \Omega(|r|n) \) where \( |r| \) denotes the number of tuples (cardinality) of table \( r \). By following Theorem 3, we get what is essentially the union among different subsets of \( r \), and the total size is \( O(|r|) \). This rule was also applied in SharedDB.

The above two corollaries show that the query optimizer can try either way of ordering \( \theta_D \) and \( \theta_1 \). Corollary 1 says that reordering the (conjunctive) terms in \( \theta_1 \) will also lead to plans of a different cost. Note that the existence of non-empty \( \Theta_i \) is very common in real-world workloads. For example, in TPC-H, all joins are done based on equality of foreign keys, as shown in the example database.

**B. Rule 2: Sequence of Projections**

In traditional DBMSs, we have

\[
\Pi_{L_1}(\Pi_{L_2}(\ldots \Pi_{L_n}(r))) \equiv \Pi_{L_n}(r)
\]

This means in a query with a sequence of projections over a relation, only the last one matters. If we consider the projections on the same relation from concurrent queries, the rule is of a different form.

**Theorem 4** (Sequence of projections). For \( n \) concurrent queries, each applies a projection over the same table \( r \), we have

\[
\Pi_{L_1}(r)\Pi_{L_2}(r)\ldots\Pi_{L_n}(r) = \Pi_{L_1'}(r)\Pi_{L_2'}(r) \ldots \Pi_{L_n'}(r)
\]

where \( L_i \) is a subset of attributes of \( r \) required by query \( q_i \).

**Proof.** We first show that, for any operand \( \Pi_{L_i}(r) \) of the LHS, all its tuples will be found in the resulting table (named \( P' \)) of RHS. While \( P' \) may drop some rows out of \( r \) due to the generation of duplicates, any duplicate in it is also a duplicate in \( \Pi_{L_i}(r) \) as \( L_i \) is a subset of the attributes of \( P' \). Given that, \( \Pi_{L_i}(r) \) can be exactly recovered from \( P' \) by simply applying the projection using its own attribute set:

\[
\Pi_{L_i}(r) = f_i(P') = \Pi_{L_i}(P')
\]

In fact, the same function \( f \) can be used to recover the operands from the resulting relation of the LHS.

**Significance:** Similar to Rule 1, we can also avoid the large size resulted from a series of \( \psi \) operations over different parts of the same table. In Rule 5 (Section III-E), we extend this idea to more general situations. Note that neither SharedDB nor DataPath applied optimization on projection like Rule 2.

**Example 4.** Referring to the example database, we write another query similar to \( q_1 \) on table PART.

\( q_5: \text{SELECT} \; \text{pkey, mfg, retprice FROM} \; \text{part} \; \text{WHERE} \; \text{mfg=m2} \)

When \( q = \{q_1, q'_1\} \), the projected attributes for each of the queries are \( L_1 = \{\text{pkey, name, retprice}\} \) and \( L'_1 = \{\text{pkey, name, mfg, retprice}\} \) then the RHS returns the table PART with \( L_2 = L_1 \cup L'_1 = \{\text{pkey, name, mfg, retprice}\} \) attributes.

**C. Rule 3: Distributive Selection**

In traditional DBMSs, selection is *distributive* over join if the selection conditions can be grouped into two parts, each of which involves attributes in only one relation, i.e.,

\[
\sigma_{\theta_1 \lor \theta_2}(T_1 \lor_c T_2) \equiv (\sigma_{\theta_1}(T_1)) \lor_c (\sigma_{\theta_2}(T_2))
\]

(9)

where \( c \) is any join condition and all the conditions in \( \theta_1 \) and \( \theta_2 \) involve attributes only in \( T_1 \) and \( T_2 \), respectively. We have similar findings regarding \( \psi \).

**Theorem 5** (Distributive Selection over \( \psi \)). With the same setup shown in Eq. (9), we have

\[
\sigma_{\theta_1 \lor \theta_2}(T_1 \psi T_2) \equiv (\sigma_{\theta_1}(T_1)) \psi (\sigma_{\theta_2}(T_2))
\]

**Proof.** First, both LHS and RHS result in relations with the same schema containing all attributes of both \( T_1 \) and \( T_2 \).
Second, we see that only tuples that pass selection condition will appear in the resulting tables of both LHS and RHS. For full outerjoin, the resulting table on both sides have three types of tuples (Fig. 2a): $T_1 \omega_2$, $T_1 T_2$, and $\omega_1 T_2$. Due to the same join conditions applied for LHS and RHS, we have the same set of tuples for all three types.

The above results can be extended to more than two tables.

**Corollary 3.** For a series of tables $T_1, T_2, \ldots, T_n$ and $\theta_i$ is a set of join conditions that only involve attributes in table $T_i$, we have

$$\sigma_{\theta_1 \theta_2 \ldots \theta_n}(T_1 \psi T_2 \psi \ldots \psi T_n) \equiv (\sigma_{\theta_1} T_1)(\sigma_{\theta_2} T_2) \psi \ldots \psi (\sigma_{\theta_n} T_n)$$

The above can be proved by reduction via proof of Theorem 5.

**Significance:** Generally, when we apply the $\psi$ operator first, we get a larger relation than if we apply the single-table selection followed by the $\psi$. This technique is popular in modern DBMSs, and was also adopted by SharedDB. In PsiDB, it can also be used to transform an intermediate expression, as discussed in Section IV-A1.

**Example 5.** In the example database, if $q = \{q_2, q_3, q_4\}$ and $\Theta = \{(\text{supprice} < \text{reprice}), (\text{supprice} < 40), (\text{avlqty} > 0), (\text{state} = \text{FL}), C_{(p, PS)}, C_{(p, S)}\}$; the selection can be distributed as

$$\sigma_{(\text{price} < \text{reprice})}(\sigma_{(\text{price} < 40)}(\sigma_{(\text{avlqty} > 0)}(\sigma_{(\text{state} = \text{FL)}C_{(p, PS)}, C_{(p, S)}))))$$

**D. Rule 4: Distributive Projection**

In traditional DBMSs, a projection is distributive over joins.

$$\Pi_{L_1 \cup \ldots \cup L_n}(T_1 \rightarrow \ldots \rightarrow T_n) \equiv \Pi_{L_1}(T_1) \rightarrow \ldots \rightarrow \Pi_{L_n}(T_n)$$

where $L_i$ is a subset of the attributes of table $T_i$. The above rule is a common query optimization strategy by filtering out attributes that will not be in the final relation $P$. In CDBMS, the following theorem shows that projection is also distributive over the $\psi$ operator.

**Theorem 6** (Distributive projection). Suppose we have $n$ concurrent queries, and query $q_i$ projects $L_i$ out of table $T_i$. If we consider an exclusive outerjoin as the $\psi$ operator, we have

$$\Pi_{L_1 \cup \ldots \cup L_n}(T_1 \psi T_2 \psi \ldots \psi T_n) \equiv \Pi_{L_1}(T_1)\psi \Pi_{L_2}(T_2) \psi \ldots \psi \Pi_{L_n}(T_n)$$

**Proof.** Firstly, both sides generate a relation with the exact same schema $L_1 \cup \ldots \cup L_n$. When we consider the exclusive outerjoin, we can see that the same set of tuples will also be generated on both sides. For such a claim, it suffices to show it for any two tables $T_1$ and $T_2$, then the results can be easily extended to $n$ tables by reduction. Specifically, for both LHS and RHS, all tuples of both $T_1$ and $T_2$ will show up in the final result. By applying the exclusive outerjoin, each tuple in $T_1$ will appear once (with $\omega_2$) for both LHS and RHS. This concludes the proof.

Eq. (10) does not hold true for an outerjoin with arbitrary conditions (i.e., inclusive outerjoins). This is because different tuples could appear in the inner join section of the final result. In particular, for the RHS of Eq. (10), the join condition can only be applied to attributes in $\cup_{i=1}^n L_i$. For the LHS, the join condition could include attributes that are not in $\cup_{i=1}^n L_i$. Therefore, the total number and content of tuples could be different. That said, we could extend the applicability of Theorem 6 as follows.

**Corollary 4.** By considering inclusive outerjoin as the $\psi$ operator, Eq. (10) holds true only if all the join conditions of LHS involve attributes that are only found in $\cup_{i=1}^n L_i$.

**Significance:** Similar to Rule 3, optimization opportunities exist when we perform the projection on individual tables first and then start the outerjoin – we avoid handling a large intermediate relation. More details of this can be found in Eq. (18) of Section IV-A1. This rule was never mentioned in SharedDB and DataPath.

**E. Rule 5: Skeleton Join**

Using $\psi$ to combine multiple queries could result in a relation that is excessively large. The following theorem shows one way to effectively control the size. Unlike previous rules, here we consider more general Selection-Projection-Join (SPJ) queries.

**Theorem 7** (Skeleton Join). Suppose we have $n$ concurrent queries, each of which involves a join among a subset of database tables $T_i \subseteq T$, i.e., $q_i = \sigma_{\theta_i}(X \neq \mathbb{T} \mathbb{E} \mathbb{T} \mathbb{R} T)$. Here $\sigma_{\theta_i}$ could contain conditions related to any table’s attributes. Let $T$ be the collection of all tables involved in the $n$ queries, i.e., $T = \cup_{i=1}^n T_i$, and $C$ be the disjunctive form of all the conditions applied in individual queries, i.e., $C = \theta_1 \lor \theta_2 \lor \ldots \lor \theta_n$, we have

$$q_1q_2\ldots q_n = \psi_{\forall T \in C} T_j$$

when $\psi_C$ is a full outerjoin with $C$ as the join condition.

**Proof.** We need to show the LHS operands are recoverable from the RHS results via two cases. Let us first define $Z = X \neq \mathbb{T} \mathbb{E} \mathbb{T} \mathbb{R} T$, which is the cross product of all tables accessed by the $n$ queries.

**Case 1:** We first consider a special case in which the same set of tables are accessed by all queries, i.e., $T_1 = T_2 = \cdots = T$. By directly applying Rule 1 (Theorem 3), we have

$$LHS = \sigma_{\theta_1}Z \sigma_{\theta_2}Z \ldots \sigma_{\theta_n}Z \equiv \sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}Z = \sigma_C Z = RHS$$

If we consider the full outerjoin as $\psi$, the resulting table of RHS, denoted as $P$, consists of two different sets: (i) all tuples in $\sigma_C Z$, in which there is no $\omega$ values; (ii) those with $\omega$ values as a result of some $T_j$ not joining with at least one other table. We can fully recover the LHS operands via the following functions:

$$f_i(P) = \sigma_{\theta_i}(P), \quad \forall i, 1 \leq i \leq n$$
Note that the function $f_i$ will return no tuples by working on subset (ii) of $P$, as the selection condition $\theta_i$ will exclude any tuple with a $\omega$ value in it. On the other hand, we can recover all tuples needed from subset (i) since $\sigma_{\theta_i}(\sigma_{C_i}Z) = \sigma_{\theta_i}Z$.

**Case 2:** Now we consider the general case in which there are queries that only access a subset of $T$, i.e., $\exists j: T_j \subset T$. For such a query $q_j$, we generate a different query

$$q_j' = q_j \times \bigotimes_{\forall \theta \in T - T_j} \omega_T$$

In other words, we concatenate each tuple in the result of $q_j$ with NULL values to generate a relation that still follows the schema of $Z$. With this transformation, we follow the same reasoning in Case 1 to show that Eq. (11) holds true if we replace $q_j$ with $q_j'$ on the LHS, meaning $q_j'$ can be recovered from the RHS results. Given $q_j'$, we can easily recover $q_j$ by projecting out the attributes that belong to $q_j$ (i.e., non-NULL attributes).

**Significance:** Rule 5 carries significant value in query optimization. First, it shows that if a table is accessed by many queries (via joins), there is no need to join it many times (according to the definition of $\psi$). The RHS of Eq. (11) is a formula in which each database table only appears once regardless of the number of queries. Since the results of every query can be recovered by the RHS of Eq. (11), we call its resulting table the **skeleton table**.

An interesting observation is that Case 1 still works if we consider cross product as the $\psi$ operator, and $\psi_C$ on the RHS is essentially an inner join, which yields a smaller resulting table as compared to using the outerjoin as $\psi$ in those cases. This is clearly an optimization path.

Case 2 in the above proof shows that, when a query accesses only a subset of the tables in $T$, we could piggyback it to the skeleton table in query processing. And the full outerjoin has to be used to recover the results of the piggybacked $q_j$ queries. Plus, we can work with the outerjoin to reduce the skeleton size as follows.

**Reducing Size of Skeleton Tables:** For $q_j$ mentioned in Case 2 of the proof of Theorem 7, there is potentially large storage overhead – in the skeleton, each tuple in $q_j$ will join with all other qualified tuples from all tables that $q_j$ does not access (i.e., those in $T - T_j$). To reduce that cost, we need to use the anti-join condition mentioned in Definition 10 to disallow such joined tuples.

With that, for aforementioned query $q_j$, we replace its join condition with $\theta_j \land \tilde{\theta}$. As a result, its tuples will not appear in the inner join segment of the skeleton table. Instead, they will only appear in the outerjoin segment in the format of $q_j'$ as mentioned above. Revisiting Eq. (11), we replace the RHS join condition $C = \bigvee_{i=1}^{m} \theta_i$ with $C = (\bigvee_i \theta_i \land \tilde{\theta})$. It is easy to see that the existence of $\tilde{\theta}$ conditions does not change the correctness of Eq. (11) – we just got a smaller skeleton table.

**Example 6.** Using the example database and queries and considering the workload of $q = \{q_1, q_2\}$, we have

$$C = \{\theta_1 \lor \theta_2\}$$

$$= ((\text{retprice} < 40 \land \tilde{\theta}) \lor (\text{supprice} < \text{retprice} \land \text{supprice} < 40 \land \text{C}_{(\text{PSA})} \land \text{C}_{(\text{PSB})}))$$

By applying Rule 5, we get an RHS of $P \psi_C \psi_{C_2} \psi_S$, by distributing the conditions in $C$ to the relevant tables. Specifically, $C_1 = ((\text{retprice} < 40 \land \tilde{\theta}) \lor (\text{supprice} < \text{retprice} \land \text{supprice} < 40 \land \text{C}_{(\text{PSA})})$ and $C_2 = \text{C}_{(\text{PSB})}$.

**F. Rule 6: Group-By and Aggregates**

Although not in the core setup of relational algebra, Grouping and Aggregates carry high practical value in analytical database systems. We therefore sketch our proposal to integrate them into the PsiDB framework and the relevant transformation rules.

**Comparison:** SharedDB proposes to handle Group-By at the tuple filtering stage, as a computation for individual queries. While adapting this idea leaves our framework for computing $P$ largely unchanged, it also misses optimization opportunities that can be captured by our following rules.

**Definition 16.** A Group-By operation over a table $r$ with the grouping criterion $B$ (a subset of attributes of $r$), denoted as $\mathcal{G}(r)$, returns an reorganized copy of $r$, in which tuples of $r$ form different groups based on the distinct values of $B$.

**Definition 17.** An aggregate $F(A_i)$ is a commutative and associative function applied to all the values in an attribute $A_i$ of a table. It is often used along with the Group-By operator, denoted as $\mathcal{G}(F(A_i))$, in which it takes all $A_i$ values in each and every group as its input. Multiple aggregates can be applied to the same Group-By result, e.g., $\mathcal{G}(F(A_i), F(A_j), \ldots F(A_k))$.

Note that Definition 17 is more general than the five aggregates (i.e., $\sum$, $\min$, $\max$, average, count) supported by SQL. The following rules can be applied to allow sharing of computation among Group-By and aggregates.

**Theorem 8** (Rule 6-1). For any table $r$, we have

$$b \mathcal{G}(r) \psi r \equiv b \mathcal{G}(r)$$

**Proof.** The Group-By (without aggregates) over a table essentially returns the same table. However, as the Group-By carries an reorganized table, we choose to keep that information on the RHS. In other words, the equation is still true if we put $r$ on the RHS.

**Theorem 9** (Rule 6-2). If $B_1 \subseteq B_2$, we have

$$b_1 \mathcal{G}(r) \psi b_2 \mathcal{G}(r) \equiv b_1 \mathcal{G}(r)$$

**Proof.** When $B_1$ is a subset of $B_2$, the groups formed by $B_2$ is of a strictly finer granularity and can be used to recover all the groups formed by $B_1$. For example, for table $Supplier$, all groups formed by $(STATE)$ can be recovered by taking groups formed by $(STATE, NAME)$.
Theorem 10 (Rule 6-3). For an aggregate function \( F(A_i) \), we denote \( F'(A_i) \) as another aggregate function that carries extra state information of size \( O(1) \) needed to compute \( F(A_i) \). If \( B_1 \subseteq B_2 \), we have

\[
B_1 \mathcal{G}(F(A_i) \cdot F'(A_i))(r) \psi B_2 \mathcal{G}(F(A_i) \cdot F'(A_i))(r) \equiv B_2 \mathcal{G}(F(A_i) \cdot F'(A_i))(r)
\] (14)

Proof. This is an extension of Rule 6-2 by considering the aggregates. It is made true by the commutativity and associativity of the aggregate functions. If we divide a dataset into multiple disjoint subsets and compute the aggregate function over all the subsets, we can compute aggregates for the whole dataset based on the partial results in constant time and space. Because \( B_1 \subseteq B_2 \), each group based on \( B_1 \) is divided into multiple groups based on \( B_2 \), therefore we can derive the aggregates of any group in \( B_1 \) giving all the results in groups of \( B_2 \). An example of \( F' \) is: for \( F \) being the average, we need either summation or count as \( F' \). □

Theorem 11 (Rule 6-4). If \( \exists b_1 \subseteq B_1 \) and \( \exists b_2 \subseteq B_2 \), and both \( b_1 \) and \( b_2 \) are candidate keys of \( r \), we have

\[
B_1 \mathcal{G}(r) \equiv B_2 \mathcal{G}(r)
\] (15)

Proof. Under this situation, \( B_1 \) and \( B_2 \) are both superkeys of \( r \) thus will generate the same groups. Therefore both sides have the same resulting table. Note that this holds true even when \( b_1 \neq b_2 \) and/or \( B_1 \neq B_2 \). □

Theorem 12 (Rule 6-5). If \( \exists b_1, b_1 \subseteq B_1 \) and \( \exists b_2, b_2 \subseteq B_2 \), plus both \( b_1 \) and \( b_2 \) are candidate keys of \( T \), we have

\[
B_1 \mathcal{G}(F(A_i))(r) \equiv B_2 \mathcal{G}(F(A_i))(r)
\] (16)

Proof. Under this situation, the same aggregate results will be obtained for both sides. However, as the schema of the resulting table contains aggregates and the grouping attributes (instead of \( r \) as in Theorem 11), the rule can only be described as weak equivalence. Note that we also have \( B_1 \mathcal{G}(F(A_i))(r) \equiv B_1 \mathcal{G}(F(A_i))(r) \).

□

Significance: The above rules capture opportunities of pushing the Group-By and Aggregates into the join skeleton (i.e., in computing \( \mathcal{P} \)) instead of treating them purely at the tuple filtering stage, as done in SharedDB. Rule 6-3 is particularly interesting: all of our other rules focused on sharing of data, but Rule 6-3 focuses on sharing of computation. In OLAP workloads with a lot of grouped computation over the same dataset [15], this can provide abundant sharing opportunities.

As a special note, when pushed to the query filters, the Group-By needs to keep a state that is of size linear to the outputs. This is caused by popular (sorting or hash-based) algorithms used for processing Group-By in. We believe this cost cannot be systematically eliminated, no matter we use CDBMS or traditional query-at-a-time systems.

IV. QUERY OPTIMIZATION IN PsiDB

We now discuss query optimization in PsiDB. We focus on algebraic optimization techniques while briefly comment on optimization at the physical plan level.

A. Algebraic Optimization Towards \( Q \)

With the introduction of full outerjoin as a cost-effective \( \psi \) operator, as well as the skeleton joins given by Theorem 7, our algebraic optimization can start from the following recommended expression for a workload.

Following the notations in Theorem 7, for \( n \) concurrent queries \( q_1, q_2, \ldots, q_n \), and each query follows the SPJ pattern, i.e., \( q_i = \Pi_{L_i} \sigma_{\theta_i} (X_{T_i \in T} T_i) \), the global query can be rewritten by starting from Rule 5 and then applying Rule 2. The expression is

\[
Q = \prod_{i=1}^{n} L_i \cup L_i(\exists T_i \in T) \]

where \( L_i \) is the set of attributes requested by query \( q_i \) in its resulting table, and \( L_i \) is the set of attributes involved in condition set \( C \). Note that the RHS of Eq. (17) is basically that of Eq. (11) with extra projections for all the attributes needed by the queries. We also need to project out the attributes in \( C \) for recovering the individual query results. It is easy to see that the projection does not change the correctness of Theorem 7 (Rule 5).

Eq. (17) serves as a starting point for query optimization in PsiDB. First, it can be made more general as follows.

Corollary 5. We can modify Eq. (17) to handle (rare) cases of a cross product between two tables being requested by a query. Note that a cross product is essentially a join with an implicit condition that always returns true. We explicitly define such a condition as \( \theta_{T_i} \) with respective to the two involved tables \( T_i \) and \( T_j \). Then we just add \( L_i \) to condition set \( C \) as a disjunctive term on the RHS.

Corollary 6. If a self-join between the same relation \( T_i \) is requested by a query, Eq. (17) is still valid upon adding another copy of \( T \) to the set \( T \) and adding the self-join condition to \( C \) as a disjunctive item.

1) Further Optimizations of \( Q \): Our equivalence rules allow us to rewrite \( Q \) to explore other optimization opportunities.

(1) According to Rule 4, for a set of attributes \( A \) that: (i) includes all attributes from table \( T_j \) that appeared in \( \bigcup_{i=1}^{n} L_i \); and (ii) includes all attributes from table \( T_j \) that appeared in \( C \), we can project out \( A \) from \( T_j \) first, i.e., we get

\[
Q = \prod_{i=1}^{n} L_i \cup L_i(\exists T_i \in T) \]

(2) Eq. (17) does not consider the order the tables are (outer) joined, as the order will not change the recoverability of the results according to the definition of \( \psi \). However, the query optimizer will evaluate different sequences of the joined tables.

(3) For the join condition \( C \), if we can locate a common component \( \theta_1 \) among all \( \theta_j \) (Definition 15) such that \( C = (\theta_j \lor \cdots \lor \theta_j) \land \theta_1 = \theta_1 \land \theta_1 \), we could follow Rule 1 (Corollary 2) to reorder the selections.

(4) Following the discussions in (3), if the common conditions among all queries \( \theta_j \) can be further written as \( \theta_j = \theta_{T_1} \land \theta_{T_2} \cdots \theta_{T_m} \), where \( \theta_{T_i} \) is a set of selection condition over
input table $T_i$, we can apply Rule 3 to push down the selection condition $\theta_i$.

2) Recovering tuples for individual queries: First of all, we assume that tuples of $P$ can be delivered to the query filters in a streaming or batching manner. This can be done by mechanisms such as cursors supported by modern DBMSs.

A recovering function will be used to filter each tuple generated from Eq. (17) or Eq. (18). Note that we have $C = \theta_1 \lor \theta_2 \lor \cdots \lor \theta_n$, therefore $q_i$ can be obtained by

$$ q_i = \Pi_{L_i} \sigma_{\theta_i}(P) \tag{19} $$

where $\mathcal{P}$ is the global query result. Another fact of interest is that the recovering function is almost identical to the query expression itself (recall $q_i = \Pi_{L_i} \sigma_{\theta_i}(\bigtimes_{T \in \mathcal{T}} T)$). This brings convenience in implementation as no extra guidance or conversion is needed for each query to recover the tuples. Now we see the reason to include $L_i$ in Eq. (17) – as each tuple arrives, we also need to apply the selection with predicates involving attributes in $\theta_i$ but not in $L_i$.

Eq. (19) shows, due to the streaming nature of selection and projection, each tuple can be processed on-the-fly and discarded afterwards. *

B. Feasibility of the PsiDB Approach

Even after the algebraic optimization, the main concern over PsiDB remains: the skeleton join in $Q$ could still be too expensive to compute, and outputs a relation that is excessively large. However, we believe such costs can be effectively controlled to an extent that it can be much lower than the total processing time of all queries in traditional systems.

1) Cost of computing $\mathcal{P}$: In Eq. (17), the skeleton join among multiple tables is scary by the first look. In particular, the condition $C$ combines many query-level conditions in a disjunctive form (i.e., $C = \theta_1 \lor \theta_2 \lor \cdots \lor \theta_n$). This could lead to scanning an entire (inner) table in processing the join. If processed in a traditional DBMS, each join becomes more expensive to compute, and outputs a relation that is excessively large. However, we believe such costs can be effectively controlled to an extent that it can be much lower than the total processing time of all queries in traditional systems.

2) Size of $\mathcal{P}$: In addition to the discussion in II-B1 about the size of the resulting relation, another concern is the size of $\mathcal{P}$ could be excessively large, thus increasing the cost of the query result filtering. This is not the case with the development of our equivalence rules, as shown in the following lemma.

Lemma 1. Every tuple in the relation $\mathcal{P}$ will appear in the resulting table of at least some query $q_i$.

*Proof. This can be shown via Eq. (17): if a tuple passed the condition $C$ from the intermediate table resulted from $\sigma_{\theta_i} T_j$, it means for some query $q_j$, its join condition $\theta_j$ is evaluated as true for that tuple, therefore the tuple will be in the final result of $q_j$. □

Denoting the cardinality of a table $T$ as $|T|$, the above lemma derives $|\mathcal{P}| \leq \sum_{i=1}^n |p_i|$. Denoting the degree and total size of a table as $D$ and $S$, respectively, we have

$$ S_{\mathcal{P}} = D_{\mathcal{P}} |\mathcal{P}| \leq \frac{D_{\mathcal{P}}}{\alpha D_p} \sum_{i=1}^n |p_i| D_p = \frac{D_{\mathcal{P}}}{\alpha D_p} \sum_{i=1}^n S_{p_i} \tag{20} $$

where $D_p$ is the average degree of the resulting tables of all queries, and $\alpha$ is the average number of queries which a tuple in $\mathcal{P}$ is used. Quantity $\alpha$ represents the overlap among the resulting tables of different queries, and Lemma 1 basically says $\alpha \geq 1$. Such overlaps could lead to significant resource savings, as shown in the SharedDB paper [7] and verified by our experiments (Section V-B2).
3) Query filtering cost: At first glance, PsiDB will be less efficient in filtering the tuples as compared to SharedDB and DataPath. PsiDB needs to broadcast each tuple to all queries while SharedDB and DataPath only sends a tuple to queries it belongs to (due to the existence of query IDs for each tuple). However, we believe the PsiDB approach will perform better for the following reasons:

   (1) We could index the predicates in the selection conditions of all queries ($\theta_i$ in Eq. (19)). Via such indexes, a tuple will be sent to only relevant queries;

   (2) For $n$ queries, each tuple in SharedDB and DataPath needs $n$ bits for the query list. Lack of such memory overhead in PsiDB allows the tuple to reside at higher levels of cache in modern CPUs/co-processors;

   (3) For each tuple, it requires $O(n)$ time to generate the query ID list in SharedDB and DataPath. Such a cost does not exist in PsiDB.

C. PsiDB Implementation

A great advantage of PsiDB is that it can be implemented on top of an existing DBMS. Note that many optimization techniques targeting efficient computation of $Q$, e.g., items (2) and (3) mentioned in Section IV-A1, are well addressed by the query optimizer of a traditional DBMS. Therefore, PsiDB can be implemented with the following levels of code disruption and depth of optimization.

Level I: we only use algebraic optimization techniques without interacting with the existing DBMS kernel. The main functionality of the query optimizer at this level is to rewrite all (declarative) queries in the workload into an expression $Q$ such as Eq. (17) or Eq. (18). The resulting expression is transformed back to a declarative query (e.g., in SQL) and sent to an existing DBMS to compute $\mathcal{P}$.

Level II: more opportunities that require significant development are considered. These include efficient outer join algorithms, optimized query batching [18], and design of the query result distributor mentioned in Section IV-A2.

V. EMPIRICAL EVALUATIONS

We accomplished a Level I implementation (Section IV-C) of PsiDB and compare it with state-of-art DBMSs. A full-fledged PsiDB system is beyond the scope of this paper. Therefore, the main purpose of our experimental study is to highlight the potential and verify key features of the PsiDB approach.

PsiDB Implementation: We implemented the PsiDB system using TSQL, the procedural language supported by SQL Server. A major component of the system is the Query Transform module, which takes a batch of queries from the Query Generator (Section V-A2) and transforms all such queries into a global expression $Q$. In particular, we apply equivalence rules to reach a level of optimization shown in Eq. (18). The query $Q$ is then transformed into an SQL statement and sent back to SQL Server to compute the final table $\mathcal{P}$. We collect the tuples of $\mathcal{P}$ in batches, and each batch is sent to an in-memory data filtering module that applies Eq. (19) to generate the individual query results. In short, this is a simple PsiDB implementation without considering any physical level optimizations.

A. Experimental Setup

1) Platform: We run all workloads in a workstation with an Intel quad-core 3.6GHz i7-7700 processor, 32GB of DDR4-2400 memory, a 256GB SSD system disk, and a 2TB 7200 RPM hard drive for database storage. All the data involved are stored on the same hard drive to ensure consistent I/O rate across different experimental runs. The workstation runs Windows 10 Enterprise Version 1709 as its OS.

2) Query Workload: We use synthetic data and query sets generated from the TPC-H benchmark for our experiments. To investigate system behavior under different data sizes, we built three databases under TPC-H scale factors (SF) 1, 10, and 100, respectively. Under SF1, the entire database was first loaded (by scanning tables) into the buffer pool thus it essentially became an in-memory database. The SF100 database is a typical disk-based system as the buffer pool can only hold a small part of the database, indexes, and runtime states. The SF10 database sits in the middle with a significant part of data and states in memory.

We developed a query generator that outputs query workloads under different numbers of tables involved, attributes to apply a selection, selectivity for selection operations, and total number of queries in a workload. In particular, we first transform all TPC-H queries into “seed” queries. The seed queries are essentially a skeleton join over multiple tables. The seed queries serve as the template for generating actual queries with selection conditions (i.e., WHERE clause in SQL) applied to randomly-picked attributes and a random subset of the attributes to be projected out (i.e., SELECT clause in SQL) from the skeleton table. The selection conditions of each attribute are randomly generated towards a desired selectivity (and thus the size of the resulting table $\mathcal{P}$). As a result, our workloads all consist of a heterogeneous set of queries with different join skeletons (Table III). Note that there are up to eight tables joined in the same query in TPC-H.

TABLE III: Weight of different query types in our workloads

<table>
<thead>
<tr>
<th>Number of joined tables</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (%)</td>
<td>10</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>13</td>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

3) Baselines: To reveal the advantages of PsiDB, we compare it with Microsoft SQL Server 13.0, which is believed to deliver excellent performance in processing a wide range of workloads [19], and the open-source MySQL 8.0. To ensure a fair comparison with baselines, we turned on the data cache and query plan cache, and also built indexes for all attributes involved in any selection condition in the queries. The baseline systems were set to use all four physical CPU cores and a buffer pool size of 24GB at all times. Such efforts are made to release the full capabilities of SQL Server and MySQL.

Unfortunately, we were not able to test other CDBMSs such as SharedDB, DataPath, and QPipe. They are either
unavailable or came as research prototypes that we were not able to compile after investing significant efforts. That said, we indirectly compare PsiDB with SharedDB and show the results in Section V-B4.

B. Experimental Results

We first report results under different workload intensity, i.e., total number of queries, database sizes, and also query selectivity. For the latter, we control the selectivity of selection conditions of the queries such that the size of table \( P \) reaches approximately 1%, 10%, 50% and 100% of the resulting table of a skeleton join (see Eq. (11)) among all eight tables in the database. This table is generated by applying all applicable foreign key equality conditions but no selection conditions over any single attribute. Therefore, this table can be viewed as the superset of data that will be retrieved by queries in a workload. We call such percentage the data coverage.

1) Overall Performance: Fig. 4 shows the speedup of PsiDB over SQL Server under different database sizes and query numbers. For the first two factors, it is easy to understand because (1) more queries; (2) higher selectivity (smaller output table size); and (3) larger database sizes. For the first two factors, it is easy to understand because the amount of computation and data accesses shared among queries increase. Note that, the case of 100% (up to 2.6X) data coverage are put here as an extreme case. Real-world workloads are likely to fall into the lower range of coverage [20]. For factor (3), when the database size increases from SF1 to SF100, it moves from an in-memory database to a disk-based database. To further investigate the effects of such a transition, we retrieved the query execution plans. We found that, in computing the global query \( Q \), SQL Server always chose table scan for every node that involves accessing a table (even though matching indexes do exist). On the other hand, when it processes queries one-by-one, index scan was used for the same. This confirms what we mentioned in Section IV-B: when system is bound by disk I/O, the scan-based algorithm shows more advantage over indexed access.
2) Cost of computing $\mathcal{P}$: By looking at the break-down of the workload processing time for PsiDB (Fig. 5), we found that the time for processing the global query $Q$ is very stable under different workload intensities. This is, again, because table scan based algorithms are chosen for processing $Q$ even under 64 queries. On the other hand, as the number of queries increases, the cost of filtering tuples from $\mathcal{P}$ increases (in a linear manner). Such costs become dominant in cases with higher number of queries. Similar trends are found in smaller databases (i.e., SF1 and SF10): the cost of computing $\mathcal{P}$ remains insignificant.

We also recorded the size of $\mathcal{P}$ and compared it with the aggregated size of all individual queries. Fig 7 shows that even under 64 queries, the quantity $\alpha = \sum |p_i|/|\mathcal{P}|$ is larger than 2.5, verifying the conclusions drawn in Lemma 1. It is also shown that $\alpha$ is positively related to the performance speedup of PsiDB over SQL Server.

3) Resource Consumption: To further study the performance of PsiDB, we collected information of runtime resource consumption for both systems. In particular, we got CPU usage (in time), Disk I/O (in pages) and Logical I/O (in pages). The CPU usage is obtained by the number of clock cycles and CPU clock speed. The Logical I/O is the total number of pages requested and Disk IO refers to the actual number of pages retrieved from disks (due to a page miss).

Fig. 8 plots the resources used by PsiDB and SQL Server under SF100 and 50% data coverage. It is very clear that the main advantage of PsiDB comes from the much lower Disk IO, which constitutes the major bottleneck in a typical database. For SF100, the number of pages loaded from memory is 8x to 419x more in SQL Server than in PsiDB, and that number is in the range of 2.6x to 176x for SF 10, and 1.8x - 45x for SF1. By looking at the CPU use, we also see a significant gap between PsiDB and SQL Server, although the gap is smaller as compared to that of the I/O.

4) (Indirect) Comparison to SharedDB: To compare PsiDB with SharedDB, we simulate the experimental environment specified in the SharedDB paper [7] and compare the speedup of both systems over the same baseline (i.e. MySQL). In particular, we generate a new set of workloads in which a query joins up to three tables in a in-memory database setup. As the selectivity of queries was on the high side (i.e., fewer tuples returned) in the SharedDB experiments, we run our experiments under 1% and 10% coverage over the skeleton table. As can be seen in Fig. 10, PsiDB achieved double-digit speedup for most cases and the highest is more than 40X. This is much higher than the 1.2-10X reported by SharedDB in similar setup (Fig. 10 in [7]). Therefore, we tend to believe the more extensive set of optimization techniques adopted in PsiDB translated into better performance (than SharedDB).

5) Adding Group By and Aggregates: In this paper, we study the potential of the PsiDB approach by focusing on SPJ-type queries. However, we also evaluated PsiDB with queries containing Group By and aggregate operators. Specifically, we following the same strategy in generating the queries except a certain percentage of the queries now carry Group By and
aggregates. Note that such a query can be viewed as an SPJ query followed by Group By and/or aggregates. Thus, the SPJ query is merged into the global query $Q$ and the latter two operations are done in the query filtering stage. Fig. 9 shows the system performance for workloads with different percentage of SPJ-only queries and those with Group By and/or aggregates. We test four scenarios with 20%, 50%, 80%, and 100% of the queries carrying Group By/Aggregates. First, PsiDB continues to beat SQL Server with Group By and aggregates in the queries. With the weight of such queries increases in the workload, the speedup decreases. Following the above discussions, this is easy to understand: the Group By and Aggregates are computationally intensive operators. Similar to workloads without Group By and Aggregates (results in Fig. 4), the impact of query batch size, database size, and data coverage on speedup does not change. For 1% coverage, the speedup still reaches 21X for SF100 and 4096 queries.

For small batch sizes that naturally have a lower demand on computation, the speedup is in the same range as in Fig. 4, which again shows the impact of computation on PsiDB performance.

6) Discussions: From the above experiments, we can clearly see the excellent performance of PsiDB as compared to mainstream DBMSs and SharedDB. The mechanism behind the success of PsiDB seems to be reduction of I/Os, which is caused by the scan-based table access method. We also want to comment on the results shown in Fig. 5. These are very interesting findings: with the increase of query numbers in the batch, PsiDB becomes a computation-bound system even under large databases (e.g., SF100). In other words, the cost of computing $P$ is insignificant, and PsiDB effectively removes the I/O bottleneck. The cost of tuple distribution/filtering becomes the dominating factor in total running time and the main reason for a low speedup under high data coverage (i.e., size of $P$). Under low data coverage, the introduction of group by and aggregates in the workload added more weight to computation and further reduced speedup of PsiDB over SQL Server. However, tuple distribution and grouping/aggregates are implemented in the most straightforward way in our PsiDB prototype, and there are abundant optimization opportunities (e.g., predicate indexing, SIMD) the current PsiDB prototype does not harness. This means the potential of PsiDB could be even bigger than what are shown here.

VI. COMPARISON TO RELATED WORK

It is well known that Codd [21] pioneered in developing the relational model formulated into an algebraic system. The
The idea of sharing execution among concurrent queries can be traced back to early reports in the context of multi-query optimization (MQO) [11]. The main idea of MQO is to keep intermediate results and reuse them for queries with the same selection conditions and eventually find a global plan for multiple queries to execute. It is shown [23] that finding the optimal global plan is an NP-hard problem. The MQO paper is the beginning of multiple-query-at-a-time system but it faces a few challenges and limitations. The main disadvantage is, by breaking the queries into conditions, it generates large amounts of intermediate results. In our framework, the intermediate data do not include any attributes unless they have been used in a query and there will not be any row that would not be used by the queries. We also focus on batching the queries that can be executed efficiently together.

Another framework for computational sharing in query processing is SharedDB [7]; it is a push-based data flow architecture for handling multiple queries as a batch. It combines the joins and decreases the amount of I/O needed by processing queries one-at-a-time. Query optimization was not explicitly studied besides a fairly simple strategy that only works with a static set of queries. Like MQO, SharedDB also does not consider the query projection. The main focus of SharedDB is combining the joins — an idea adopted from CJoin [1]. In our framework we only return data that will be needed in the queries’ final results. The middle table is pruned by the selection and projection of all the queries. Eventually the result of each query will be distributed and processed on-the-fly.

The PsiDB framework is in close spirit to SharedDB (except for several aspects that we discussed in details). The main contribution of our paper lies in the formal reasoning of concurrent query processing at the level of relational algebra. We develop equivalence rules to guide the creation of query plans for processing a batch of queries together. Therefore, unlike SharedDB that currently focuses on one static plan, our work will pave the way for a systematic consideration of optimizing multiple queries in CDBMSs.

Another group of work focus on sharing the I/O resources. CJoin [1], Datapath [6], and QPipe [5] are three such systems. A key feature of these systems is that they rely on temporal overlap for sharing. In such systems, whole table scans are often found to outperform index-based scans (discussed in [17]). With multiple queries asking for the same table being scanned, the data will be shared among all of them. The scan sharing techniques are also implemented in modern database systems. MonetDB [24] optimizes disk bandwidth utilization via cooperative scans where real-time scheduling queries can be performed according to their data requests and the current status of the buffers. Similarly, Blink [25] and Crescendo [26] process multiple queries in one table scan to save disk bandwidth and memory.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce a framework for multiple query processing via query batching. We develop relational algebraic equivalence rules to transform multiple queries into a single global query. The results of the global query are distributed among all the queries on-the-fly. We lay the foundation for such a framework by showing the abundant optimization opportunities that are captured by our system model and equivalence rules. We then recommend initial algebraic plans for query optimization, evaluate feasibility of the framework in comparison to similar systems. Experiments run on an early prototype of our system demonstrate great potential of our approach.

This paper leaves many open problems that are worth immediate explorations. On the theoretical side, development of more equivalence rules to reveal other resource sharing scenarios is beneficial, especially for non-algebraic operators such as Group-By and Aggregates. A more fundamental work is to model the total cost of processing the large number of queries in the PsiDB manner. This is much needed to reveal the actual performance advantage of scanning-based data processing system in large. The problem of optimal batching of queries is obviously a critical issue for the proposed PsiDB framework, deeper understanding of the problem structure and features is needed for the development of efficient algorithms. Heuristic solutions with guaranteed performance is the way to pursue in such studies. As we mentioned in the paper, design and implementation of a query optimizer following the PsiDB strategy will be our next topic. While PsiDB can be implemented with existing query optimizers, there are interesting problems such as efficient processing of the $\psi$ operators, parallel computing for query filtering, and cost modeling of query plans.

REFERENCES

We start by studying the difference between two associative forms among three tables $T_1$, $T_2$ and $T_3$, as illustrated in Fig. 11. Here $\omega_1$ represents a tuple following the schema of $T_1$ but with NULL values in all attributes. Note that the tuples in the resulting relation of either associative form should come from the skeleton relation $S = T_1^c \times T_2^c \times T_3^c$ where $T_i^c = T_i \cup \omega_i$. In particular, a series of conditions are applied to select tuples out of $S$. Without loss of generality, such conditions are shown as:

1. those between tables $T_1$ and $T_2$ only;
2. those between tables $T_2$ and $T_3$ only;
3. those between tables $T_1$ and $T_3$ only.

Note that any condition that involves attributes from only one table is irrelevant to our discussions thus are not shown here. Now we consider the seven different types of tuples in the resulting table of either form. Such types differ by the appearance of $\omega$ segments. For example, tuples of type $T_1 T_2 \omega_3$ consist of a real tuple from $T_1$, a real tuple from $T_2$, and NULL values in all attributes of the $T_3$ section. By analyzing the order of outer join operations between tables in both associative forms, we can derive how the three sets of conditions are evaluated in each type of tuples, as shown in Fig. 11. For example, to get the $T_1 T_2 \omega_3$ tuples, we must have conditions 1 evaluated as true and 3 as false in the left formula. For the right formula, such tuples are generated as a result of conditions 1 being true plus either 2 or 3 is false. Note that for tuple types $T_1\omega_2T_3$ for the left form and $\omega_1\omega_2T_3$ on the right form, each is generated under three different scenarios.

![Fig. 11: Two forms of consecutive full outer joins over three tables and the seven types of tuples in them](image-url)

Clearly, the two forms are not equivalent - multiple tuple types have different evaluations of all conditions thus contain different sets of tuples. However, we only need to show that all three tables can be exactly recovered from both sides.

We show that by applying the recovering function shown in Eq. (2) to the resulting table of both forms, we can get all three input tables back. Recall the function to recover $T_i$ is:

$$f(\mathcal{P}) = \sigma_{\exists k, \#null_1 \geq 1}(\mathcal{P})$$

For that, we can gather all the relevant components for a table shown in Fig. 11. For example, all components needed for
recovering \( T_2 \) is shown in Fig. 12. The set of \( T_2 \) tuples that can be recovered on the left form can be obtained by

\[
f(i) \cup f(ii) \cup f(iii) \cup f(iv)
\]

In each component, an associated condition evaluation defines a subset of the \( T_2 \) tuples. By working on all four components, the total recoverable set for \( T_2 \) can be computed as a selection over \( T_2 \) with the following conditions:

\[
\begin{align*}
(2+ & \land 1+ \land 3+) \lor (2+ \land (1- \lor 3-)) \\
& \lor (2- \land 1+) \lor (2- \land 1-) \\
& = 2+ \land (1+ \land 3+) \lor (1- \lor 3-) \\
& \lor (2- \land 1+) \\
& = (2+ \land TRUE) \lor (2- \land TRUE) = TRUE
\end{align*}
\]

This means that all tuples of \( T_2 \) will be recovered by \( f \).

Similarly, for the right form, we get the set of \( T_2 \) tuples via \( f(iv') \lor f(iii') \lor f(ii') \lor f(i') \), which is a select over \( T_2 \) by the following conditions:

\[
\begin{align*}
(1+ & \land 2+ \land 3+) \lor (1+ \land (2- \lor 3-)) \\
& \lor (1- \land 3+) \lor (1- \land 2+) \\
& = (1+ \land (2+ \land 3+) \lor (2- \lor 3-) \\
& \lor (1- \land (2- \lor 3+)) \\
& = (1+ \land TRUE) \lor (1- \land TRUE) = TRUE
\end{align*}
\]

\[\begin{array}{c|c}
\hline
\text{Component} & \text{Condition} \\
\hline
i & T_1, T_2, T_3 \quad 1+ \land 2+ \land 3+ \\
ii & T_1, T_2, \omega_3 \quad 1+ \land 2+ \\
iii & \omega_1, T_2, \omega_3 \quad 1- \land 3- \\
iv & \omega_1, T_2, T_3 \quad (1- \lor 3-)
\end{array}\]

\[\begin{array}{c|c}
\hline
\text{Component} & \text{Condition} \\
\hline
i' & T_1, T_2, T_3 \quad 1+ \land 2+ \land 3+ \\
ii' & T_1, T_2, \omega_3 \quad 1+ \land (2- \lor 3-) \\
iii' & \omega_1, T_2, \omega_3 \quad 1- \land 3- \\
iv' & \omega_1, T_2, T_3 \quad (1- \lor 3+)
\end{array}\]

Fig. 12: Components needed for recovering table \( T_2 \) from the two resulting tables shown in Fig. 11

The same results can be found for \( T_1 \) and \( T_3 \), we skip the tedious details here and conclude the proof.

**APPENDIX B**

**MORE EXPERIMENTAL RESULTS**

Such results can be found in Figs. 16 to 15.
Fig. 16: Execution time of PsiDB and SQL Server under different database sizes and query numbers