PsiDB: A Framework for Batched Query Processing and Optimization

Mehrad Eslami
University of South Florida

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.
PsiDB: A Framework for Batched Query Processing and Optimization

by

Mehrad Eslami

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yi-Cheng Tu, Ph.D.
Sriram Chellappan, Ph.D.
Xinming Ou, Ph.D.
Mingyang Li, Ph.D.
Ming Ji, Ph.D.

Date of Approval:
December 06, 2019

Keywords: Batch processing, Batch partitioning, Query optimization, Equivalence rules

Copyright © 2019, Mehrad Eslami
# Table of Contents

List of Tables iv  
List of Figures v  
Abstract vii  

Chapter 1: Introduction 1  

Chapter 2: Related Work 4  
2.1 Background 4  
2.1.1 Single-query-at-a-time Processing 4  
2.1.2 Multiple-query-at-a-time Processing 7  
2.1.2.1 Shared Sub-Expressions and Execution 7  
2.1.2.2 Shared Query Execution 8  
2.1.2.3 Shared Scan 9  
2.2 Existing Batch Processing Databases 9  

Chapter 3: PsiDB Framework 12  
3.1 The PsiDB Framework 12  
3.2 Multi-Query-One-Execution Model 12  
3.3 Combining Multiple Relations 14  
3.3.1 Size of Resulting Relation 21  
3.4 Filtering the Global Query 22  

Chapter 4: PsiDB Optimization 24  
4.1 Equivalence Rules in PsiDB 24  
4.2 Rule 1: Sequence of Selections 26  
4.3 Rule 2: Sequence of Projections 29  
4.4 Rule 3: Distributive Selection 30  
4.5 Rule 4: Distributive Projection 31
4.6 Rule 5: Skeleton Join ................................................. 33
4.7 Rule 6: Group-By and Aggregates ................................. 36
4.8 Query Optimization in PsiDB .................................... 38
   4.8.1 Algebraic Optimization Towards Q ......................... 39
   4.8.2 Further Optimization of Q ................................. 41
   4.8.3 Recovering Tuples for Individual Queries ................. 42
4.9 Feasibility of the PsiDB Approach .............................. 43
   4.9.1 Cost of Computing P ........................................ 43
   4.9.2 Size of P .................................................. 45
   4.9.3 Query Filtering Cost ...................................... 46
4.10 PsiDB Implementation ........................................... 47

Chapter 5: Batch Partitioning .......................... 48
   5.1 A Research Gap ................................................. 48
   5.2 Problem Description .......................................... 50
   5.3 The Proposed Approach ..................................... 52
      5.3.1 The Batch Processing Time Prediction Function ....... 52
      5.3.2 The Optimization Model ................................. 55
         5.3.2.1 Computational Complexity ......................... 58
      5.3.3 Heuristic Solution Methods ............................ 60

Chapter 6: PsiDB Experimental Results .............. 63
   6.1 PsiDB Evaluation .............................................. 63
      6.1.1 Experimental Setup .................................... 63
         6.1.1.1 Platform ........................................... 63
         6.1.1.2 Query Workload .................................. 64
         6.1.1.3 Baselines ......................................... 65
      6.1.2 Experimental Results .................................. 65
         6.1.2.1 Overall Performance ............................... 66
         6.1.2.2 Lower Bound ...................................... 67
         6.1.2.3 Cost of Computing P ............................. 70
         6.1.2.4 Resource Consumption ............................ 70
      6.1.3 More Experimental Results ............................ 71
         6.1.3.1 (Indirect) Comparison to SharedDB .............. 71
         6.1.3.2 Adding Group By and Aggregates ................. 72
         6.1.3.3 Discussions ........................................ 73
   6.2 A Computational Study on Partitioning Batches .......... 73
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Queries in sample workload</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>The tables and their relations of Figure 4.2</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Weight of different query types in our workloads</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>The number of records in each table of TPC-H for three values of SF</td>
<td>74</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of RCSA-I and RCSA-II on TPC-H databases</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparing the quality of the solutions obtained by the proposed heuristics with optimal solutions</td>
<td>78</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of RCSA-I and RCSA-II on the JOB database</td>
<td>79</td>
</tr>
<tr>
<td>6.6</td>
<td>Comparing the quality of the solutions obtained by the proposed heuristics with optimal solutions</td>
<td>81</td>
</tr>
<tr>
<td>Figure Number</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1.1</td>
<td>A relational database management system components</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>The PsiDB concurrent query processing system model</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>An illustration of the resulting relation of a full outerjoin between two tables $T_1$ and $T_2$</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Two forms of consecutive full outer joins over three tables and the seven types of tuples in them</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Components needed for recovering table $T_2$ from the two resulting tables shown in Fig. 3.2</td>
<td>20</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Caption</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Speedup of three queries via sequential scan over that by index-based access.</td>
<td>45</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>A single batch-processing, query optimizer model</td>
<td>49</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>A batch-processing model with partitioned query</td>
<td>50</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Speedup of PsiDB over SQL Server under different database sizes and query numbers</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Time for processing $Q$ and tuple distribution under SF100</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>$(\sum</td>
<td>p_i</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Total resource consumption of PsiDB and SQL Server in running workloads SF1, SF10 and SF100 and 50% data coverage</td>
<td>68</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Speedup of PsiDB over SQL Server with batches including queries with group by and aggregates</td>
<td>69</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Speedup of PsiDB over MySQL</td>
<td>71</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Time for processing $Q$ and distribution under (a) SF=1, (b) SF=10, and (c) SF=100</td>
<td>82</td>
</tr>
</tbody>
</table>
Figure 6.8: Execution time of PsiDB and SQL Server under different database sizes and query numbers ........................................... 83
Figure 6.9: The structure of a TPC-H database ........................................... 83
Figure 6.10: Total time (in seconds) of all 20 instances of the smallest and largest subclasses (white bars: default setting, gray bars: RCSA-I, black bars: RCSA-II) ........................................... 84
Figure 6.11: The structure of the JOB database ........................................... 85
Figure 6.12: Total time (in seconds) of all 20 instances of different classes (white bars: default setting, gray bars: RCSA-I, black bars: RCSA-II) ........................................... 86
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 50X over a mainstream commercial RDBMS.

Following the development of the system and showing how effective they are in handling large number of queries, there is a lack of rigorous modeling and theoretical study for query batching. Query batching in database systems has strong resemblance to order batching in the warehousing context. While the latter is a well-studied problem, the literature on optimization techniques for query batching problem is quite scarce/nonexistent. In following of PsiDB framework, we develop a Mixed Binary Quadratic Program (MBQP) to optimize query-batching in a database system. This model uses the coefficients of a linear regression on the query retrieval time, trained by a large set of randomly generated query sets. We also propose two heuristics, for solving the proposed MBQP. To demonstrate the effectiveness of our proposed techniques, we conduct a comprehensive
computational study over two well-known database benchmarks. The computational results show that when the proposed MBQP is solved using the designed heuristics, an improvement of up to 45% in retrieving time over a single batch is achieved.
Chapter 1: 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, 2, 3, 4, 5, 6, 7]. Although such work (reviewed in Section 2) comes with different flavors, in this research, we call them concurrent database management systems (CDBMS).

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 [7, 5, 8, 6], 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 speed up of 1.4x in our worst case scenario, over one-query-at-a-time by SQL Server.

Dissertation Organization: In Chapter 2, we review related work in this field; in Chapter 3, we introduce the PsiDB framework for batched query processing; in Chapter 4, we present a series of equivalence rules geared towards concurrent query processing and we show key challenges in the design of a PsiDB query optimizer; in Chapter 5 we present a batch partitioning system for better batching the queries and improving PsiDB performance; and we presents empirical evaluation results in Chapter 6; and we conclude this dissertation and comment on future directions at the end of Chapter 6.
Chapter 2: Related Work

2.1 Background

Query Processing in traditional one-query-at-a-time DBMSs is one of the most heavily studied topics in database systems. A good survey of this topic can be found in [9]. The query processing time relies on access path cost to data and processing time. There are two major qualities that distinguish the database systems, first the ability to manage persistent data and second the capability to access large amounts of data efficiently. The resource utilization makes that efficiency to access the large data happen and it highly focuses on the query selectivity and the amount of data we need to access and process to obtain the results. A large number of studies have worked on selectivity estimation [10, 11, 12, 13]. It is well known that Codd [14] pioneered in developing the relational model formulated into an algebraic system. The basic mathematical foundation of relations, however, should be accredited to De Morgan [15] in the 1860s. The algebraic transformations are generally regarded as the first step in query optimization. With relational algebra, it is possible to rearrange the operations to get alternative access paths with different execution costs [16]. In particular, an equivalence rule in relational algebra transforms an algebraic statement to a different yet logically equivalent form. This is the foundation of query optimization in many papers that proposed different methods to increase the query processing efficiency [17, 18].

2.1.1 Single-query-at-a-time Processing

Since in this research we mainly focus on relational databases, it is essential to have fundamental knowledge about databases for those who does not have a database background. Hence,
we survey a few key components of relational databases briefly in this section. The system software that handles the database functionality is called a Database Management System (DBMS). An RDBMS is a type of DBMS which follows certain rules designed by [14] using the relational model. In the following, we explain the structure of data in relational databases and the methods of retrieving data efficiently from such a database. Figure 1.1 shows the main components of an RDBMS. We briefly explain each component in this section.

**Data Structure:** Data values in an RDBMS are stored in the form of tables. A table is a fixed data structured that consists of columns (also called attributes) and rows (Tuples). A table has a specified number of columns, but can have any number of rows. Each row is a record stored in the database and each column is a type of data whose domain (e.g., data type, length, range of value) is pre-described. In designing a database table, certain attribute(s) of a table are set to be unique identifiers of the entire row, and such attribute(s) is called a key. There can be multiple keys in a table, e.g., Social Security Number (SSN) and Student ID are both keys in a University Student table as they both can be viewed as IDs of students (rows). Typically there are two kinds of key: *Primary key*, which is chosen among all keys in a table by the database designer, and *foreign keys*, referring to another table’s primary key. The computations in such systems are described by an algebraic system named *Relational Algebra* (RA). Among the RA operators, concatenating rows from different tables is called *join.*
**Queries:** A query is a language expression that describes data to be retrieved from a database based on specific criteria. All the queries in an RDBMS are in Structured Query Language (SQL) form. The SQL language has changed through the time and deviated in several ways from its theoretical foundation. Our focus in this paper is on standard query expression of SQL. Such expressions are done in a *declarative* manner, which focuses on descriptions of the query results. A simple SQL statement (i.e. `SELECT a_1, ..., a_w FROM t_1, ..., t_m WHERE c_1, ..., c_f`) has the following parts:

- **SELECT:** Specifies the columns in the result
- **FROM:** The table(s) which data is stored in
- **WHERE:** A set of conditions which specifies criteria for the retrieved rows

The scope of this study is on the queries of the form `SELECT . . . FROM . . . WHERE` (SFW). Such queries are known to be one-to-one mappings to core operators in relational algebra, which is the query language used to described actual computations required to process the queries.

**SQL Parser:** SQL parser handles two tasks: Syntax validations and query transformation. Any query that is being executed, needs to be translated to the relational algebraic statement first. We have the same main components in the relational algebraic:

- **Projection:** The set of columns/attributes written as $\Pi_{a_1, \ldots, a_w}$;
- **Join:** The tables needed to execute the queries;
- **Selection:** The specific conditions on the attributes $\sigma_{c_1, \ldots, c_f}$.

A SFW query in SQL language, translates to $\Pi_{a_1, \ldots, a_w} \sigma_{c_1, \ldots, c_f}(t_1 \times \ldots \times t_m)$ in algebraic form. After this transformation, the query is ready to be optimized and then executed.

**Query Optimizer:** The query optimizer is a main component in terms of database performance enhancement. The query optimizer attempts to determine the most efficient way to execute a query.
by creating multiple execution plans and choose the optimal one. This happens right after parsing the query and before accessing the data. A query plan determines the query execution plan and how to access the data. Two different plans for the same query would have the same results but the execution time and resource consumption for each of them can be different.

**SQL Executor:** Given a query execution plan, the query processor fetches the data in the tables and generates the results. There are many factors that play major roles in a query execution such as buffering or sequential reading, which are outside the scope of this paper.

### 2.1.2 Multiple-query-at-a-time Processing

The idea of sharing execution among concurrent queries can be traced back to early reports in the context of multi-query optimization (MQO) [20]. The main idea of MQO is to find a global plan for multiple queries to execute them together. In [20], two optimizers, one local and one global optimizer, are proposed. The local optimizer estimates cost of a query and finds an optimized plan for that query. In the next step, the global optimizer creates a global plan over all the individual query plans towards minimum total cost. The global plan is generated by a heuristic algorithm. In another paper [21], the same group of authors show that finding the optimal global plan is an NP-hard problem. Another approach of multiple query processing is to increase the chance of resource sharing among different queries. The basic units for sharing can be sub-queries, common expression, intermediate results, and loaded data. In particular, such work fall into the following categories.

#### 2.1.2.1 Shared Sub-Expressions and Execution

There are many reports on detection of common sub-expressions, and finding the common query parts as well as shared nested queries in purpose of multi-query optimization [6, 22, 23]. An issue with this method is that the performance relies on high concurrency among the queries and the
design of heuristic algorithms for merging the plans. When the queries are broken down to multiple sub-queries with selection conditions that do not contain each other, the system end up restoring entire data. Another issue is related to optimizer performance matters. Commonly, with increasing the number of query and complexity of queries, the search space for the optimizer increases as well. It leads to longer searches to find alternative plans. Some research proposed ignoring the alternative plans makes the system not efficient on one query but increase efficiency overall. A sample of those frameworks is SharedDB [7], which uses a push-based data flow architecture. The query optimizer does not necessarily adopt the best query plan for each run, instead it aims at decreasing the computational time and ultimately improving the average query response time. SharedDB batches the queries and creates a static plan for a database schema to perform the query processing. This idea was adopted from CJoin [1], which introduces a system for sharing data to operate joins by a static query execution plan.

2.1.2.2 Shared Query Execution

In other respects, the execution sharing mechanism detects and jointly executes common sub-expressions in a given set of queries and restore the results to be able to use them again in future. One common technique in order to save resources is, the queries are broken into multiple sub-queries with different selection conditions and the intermediate results are stored in the database for further retrievals [20]. Thereupon the algorithm optimizes \( n \) queries and finds a global plan by calculating the costs of each local plan as well as merging them. A main drawback in sharing execution would be the storage overhead. It is also difficult to choose what data/results to store due to the lack of reliable algorithms for predicting future trends. Grupcev and co-workers [24] utilizes a similar idea in data analytical workloads of scientific databases. They make the observation that a limited number of data primitives are frequently used as building blocks for more complex analytics, therefore such primitives only need to be computed once.
2.1.2.3 Shared Scan

A typical query-centric database management system incorporates a buffer pool and employs eviction policies. The buffer management does the critical job of feeding the queries with the needed data [25, 26, 27]. However, with the volume of data growing faster than that of available memory, I/O cost continues to dominate the query processing time and thus the main target for query optimization. Recent years have seen a movement of designing DBMSs with I/O sharing mechanisms. CJoin [1] and Datapath [6], and QPipe [5] are three such systems that use a simultaneous pipelining technique to share the execution of queries that arrive within a certain time frame. 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 as the former can thinly spread the I/O cost to all concurrent data processing tasks and minimize random I/O. Qpipe is one of sharing system that is designed to maximize sharing of data across concurrent queries by pipelining the operator’s outputs to multiple parent nodes. A micro-engine is implemented for each operator. The ultimate goal of Qpipe is to increase the data processing throughput for multiple queries. The advantage of table scan over index-based scan is confirmed in a recent work [28]. The scan sharing techniques are also implemented in modern database systems. MonetDB [29] 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 [30] and Crescando [31] process multiple queries in one table scan, independent of the query predicates, thus the disk bandwidth and memory are shared. In those systems, the degree of sharing depends on number of concurrent queries and commonalities among results.

2.2 Existing Batch Processing Databases

The idea of sharing execution among concurrent queries can be traced back to early reports in the context of multi-query optimization (MQO) [20]. The main idea of MQO is to keep interme-
diate 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 [21] 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 IO 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 [28]). 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 [29] 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 [30] and Crescando [31] process multiple queries in one table scan to save disk bandwidth and memory.
Chapter 3: PsiDB Framework

3.1 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.

3.2 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 $R$. 

12
**Definition 2** A query \( q_i \) 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 \( A_k \). The resulting relation of query \( q_i \) contains a set of attributes \( L_i = \{A_1, A_2, ..., A_k\} \). This set \( L_i \) 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 \( L_q = L_1 \cup L_2 ... \cup L_n \).

A database contains a set of relations \( \mathbb{T} = \{T_1, T_2, ..., T_m\} \), 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 \( \mathbb{T} \) 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 [32].

**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 3.1.

In a CDBMS, an expression should return a vector of relations (vs. one single relation in RDBMS) 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
### Table 3.1: Queries in sample workload

<table>
<thead>
<tr>
<th>Query</th>
<th>SQL Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td><code>SELECT pkey, P.name, retprice</code>&lt;br&gt;<code>FROM part P WHERE retprice &lt; 40</code></td>
</tr>
<tr>
<td>q₂</td>
<td><code>SELECT pkey, MIN(retprice), MAX(supprice)</code>&lt;br&gt;<code>FROM part P JOIN partsupp PS JOIN supplier S</code>&lt;br&gt;<code>ON P.pkey = PS.pkey AND S.skey = PS.skey</code>&lt;br&gt;<code>WHERE supprice &lt; retprice AND supprice &lt; 40</code>&lt;br&gt;<code>GROUP BY pkey</code></td>
</tr>
<tr>
<td>q₃</td>
<td><code>SELECT pkey, skey, retprice, supprice, S.name</code>&lt;br&gt;<code>FROM part P JOIN partsupp PS JOIN supplier S</code>&lt;br&gt;<code>ON P.pkey = PS.pkey AND S.skey = PS.skey</code>&lt;br&gt;<code>WHERE supprice &lt; retprice AND avlqty &gt; 0</code></td>
</tr>
<tr>
<td>q₄</td>
<td><code>SELECT state, SUM(supprice)</code>&lt;br&gt;<code>FROM part P JOIN partsupp PS JOIN supplier S</code>&lt;br&gt;<code>ON P.pkey = PS.pkey AND S.skey = PS.skey</code>&lt;br&gt;<code>WHERE supprice &lt; retprice AND state = FL</code>&lt;br&gt;<code>GROUP BY state</code></td>
</tr>
<tr>
<td>q₅</td>
<td><code>SELECT skey, pkey, state</code>&lt;br&gt;<code>FROM supplier S</code>&lt;br&gt;<code>JOIN partsupp PS ON S.skey = PS.skey</code>&lt;br&gt;<code>WHERE state = FL</code></td>
</tr>
</tbody>
</table>

Each tuple in \( P \) will then be distributed to filter functions that belong to individual queries to generate the final result for the latter. Fig. 5.1 illustrates the PsiDB system design, and we elaborate on key components of the model as follows.

### 3.3 Combining Multiple Relations

With the concept of a global relation \( P \), it is necessary to develop an approach to combine results of individual relational expressions into \( P \). Specifically, we need to define a binary relational operator for such purposes. Intuitively, such an operator: (i) should generate a global output relation \( P \) 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 \( P \). It turns out more than one relational operators satisfy such conditions. Let us first formalize the definition of such operators.
Note that the common concepts of *commutativity* (i.e., \( T_1 \circ T_2 = T_2 \circ T_1 \)) and *associativity* (i.e., \( T_1 \circ (T_2 \circ T_3) = (T_1 \circ T_2) \circ T_3 \)) 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

\[
T_1 \times T_2 = \{<t_1, t_2> | \forall t_1 \in T_1 \wedge \forall t_2 \in T_2\}
\]

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

\[
T_1 = \Pi_{L_{T_1}}(\mathcal{P}) \quad , \quad T_2 = \Pi_{L_{T_2}}(\mathcal{P})
\]

(3.1)

*Commutativity and Associativity:* It is easy to see that cross product is (strong) commutative and (strong) associative. Detailed proof can be found in [33, page 79].

\( \Box \)

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

**Definition 4** Selection condition \( \theta \) on a relation \( r \) compares two atomic values of \( \theta = A_i \phi A_j \) or \( \theta = A_i \phi e \) where \( \phi = \{<, =, \neq, >, in\} \) and \( e \) is a valid value for the attribute \( A_i \).
Definition 5 A selection operation over a relation \( r \) is to apply a set of conditions as follows:
\[ \sigma_\theta(r) = \{t|t \in r \land \theta(t) = TRUE\} \].

Definition 6 The (inner) join between two relations by a common condition \( c \) is
\[ T_1 \bowtie_c T_2 = \sigma_c(T_1 \times T_2) \].

Definition 7 A left outer join between two relations is denoted as \( T_1 \bowtie_L c T_2 \) where \( c \) is the join condition. Let \( L_{T_1} \) be the attribute list of table \( T_1 \) and let \( \omega_2 = \{(null, \ldots, null)\} \) be the singleton on all attributes of \( T_2 \), the left outer join can be described as follows:
\[ T_1 \bowtie_L c T_2 = (T_1 \bowtie_c T_2) \cup \left((T_1 - \Pi_{L_{T_1}}(T_1 \bowtie_c T_2)) \times \omega_2\right) \].
Similarly, we have the right outer join defined as:
\[ T_1 \bowtie_R c T_2 = (T_1 \bowtie_c T_2) \cup \left((T_2 - \Pi_{L_{T_2}}(T_1 \bowtie_c T_2)) \times \omega_1\right) \].

Definition 8 A full outerjoin is a union between both left and right outerjoin (definition 7) on the same relations:
\[ T_1 \bowtie c T_2 = (T_1 \bowtie_L c T_2) \cup (T_1 \bowtie_R c T_2) \].

Theorem 2 The full outer join (\( \bowtie \)) is a \( \psi \)-operator.

Proof: We consider \( \mathcal{P} = T_1 \bowtie c T_2 \) with any general join conditions \( c \) (Fig. 3.1a).

Operand Recoverability: The resulting table of a full outer join 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

$$T_1 = \sigma_{\forall A_i \neq \text{null}} \Pi_{L_{T_1}}(P), \quad T_2 = \sigma_{\forall A_i \neq \text{null}} \Pi_{L_{T_2}}(P)$$

(3.2)

where $A_i$ 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. 3.1a).

**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 [33]; 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 [33, page 96], the full outerjoin is strong associative with certain assumptions made with respect to the join conditions. In following we prove that it is weak associative under any general join conditions.

We start by studying the difference between two associative forms among three tables $T_1$, $T_2$ and $T_3$, as illustrated in Fig. 3.2. Here $\omega_i$ represents a tuple following the schema of $T_i$ 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 $S = T'_1 \times T'_2 \times T'_3$ where $T'_i = T_i \cup \omega_i$. 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 $T_1T_2\omega_3$ 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. 3.2. For example, to get the $T_1T_2\omega_3$ tuples, we must have conditions $\circ$ evaluated as true and $\circ$ as false in the left formula. For the right formula, such tuples are generated as a result of conditions $\circ$ being true plus either $\circ$ or $\circ$ is false. Note that for tuple types $T_1\omega_2\omega_3$ for the left form and $\omega_1\omega_2T_3$ on the right form, each is generated under three different scenarios.

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. (3.2) to the resulting table of both forms, we can get all three input tables back. Recall the function to recover $T_i$ is

$$f(P) = \sigma_{\forall A_i \neq \text{null}} \Pi_{T_i}(P)$$

For that, we can gather all the relevant components for a table shown in Fig. 3.2. For example, all components needed for recovering $T_2$ is shown in Fig. 3.3. The set of $T_2$ tuples that can be
Figure 3.2: Two forms of consecutive full outer joins over three tables and the seven types of tuples in them

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:

\[(\text{2}^+ \land \text{1}^+ \land \text{3}^+) \lor (\text{2}^+ \land (\text{1}^- \lor \text{3}^-)) \lor (\text{2}^- \land \text{1}^+) \lor (\text{2}^- \land \text{1}^-)\]

\[= \text{2}^+ \land ((\text{1}^+ \land \text{3}^+ \lor \text{1}^- \lor \text{3}^-)) \lor (\text{2}^- \land (\text{1}^- \lor \text{1}^+))\]

\[= (\text{2}^+ \land \text{TRUE}) \lor (\text{2}^- \land \text{TRUE}) = \text{TRUE}\]

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 \( f(i') \cup f(ii') \cup f(iii') \cup f(iv') \), which is a select over \( T_2 \) by the following conditions:

\[(\text{1}^+ \land \text{2}^+ \land \text{3}^+) \lor (\text{1}^+ \land (\text{2}^- \lor \text{3}^-)) \lor (\text{1}^- \land \text{2}^-) \lor (\text{1}^- \land \text{2}^+)\]

\[= (\text{1}^+ \land (\text{2}^+ \land \text{3}^+ \lor \text{2}^- \lor \text{3}^-)) \lor (\text{1}^- \land (\text{2}^- \lor \text{2}^+))\]

\[= (\text{1}^+ \land \text{TRUE}) \lor (\text{1}^- \land \text{TRUE}) = \text{TRUE}\]

\[\square\]

Figure 3.3: Components needed for recovering table \( T_2 \) from the two resulting tables shown in Fig. 3.2
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 (∪) operator. However, this excludes operands that are not union compatible and thus is not as general a solution as ψ. Various multi-query-optimization (MQO) approaches [20] focus on combining common subexpressions among queries while PsiDB works on entire query expressions. Certainly, there are practical issues with the use of ψ, one major concern being the size of its output table. We will develop various techniques to address such a challenge.

3.3.1 Size of Resulting Relation

With the ψ-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. 3.1a). To minimize such redundancy, we could apply a dummy condition to disallow any joined tuples.

**Definition 9** Given two tables \( T_1 \) and \( T_2 \), an anti-join condition \( \bar{\theta} \) ensures \( T_1 \bowtie_{\bar{\theta}} T_2 = \emptyset \), and \( T_1 \bowtie_{\bar{\theta}} T_2 = T_1 \times \omega_2 \).

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 10** 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 [34, 35]. 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. 3.1b).

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 4.1) in query optimization.

### 3.4 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 4.8.3).

**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 multi-casted 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 4.8.3.
Chapter 4: PsiDB Optimization

4.1 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 [36]. 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 11** Suppose a relational algebraic expression $E_1$ contains a series of $\psi$ operations 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 $E_1 \simeq E_2$, 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 12 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 $\Theta_T = \Theta_1 \cup \Theta_2 \cup \ldots \cup \Theta_n$.

Example 2 The conditions sets for query workload as shown in Table 3.1 are as follows.

$$\Theta_1 = \{(\text{retprice} < 40)\}$$
$$\Theta_2 = \{(\text{supprice} < \text{retprice}), (\text{supprice} < 40), (C_{(S,PS)}), (C_{(PS,P)})\}$$
$$\Theta_3 = \{(\text{supprice} < \text{retprice}), (\text{avlqty} > 0), (C_{(S,PS)}), (C_{(PS,P)})\}$$
$$\Theta_4 = \{(\text{supprice} < \text{retprice}), (\text{state} = \text{FL}), (C_{(S,PS)}), (C_{(PS,P)})\}$$
$$\Theta_5 = \{(\text{state} = \text{FL}), (C_{(PS,S)})\}$$

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

$$\Theta_T = \{(\text{retprice} < 40), (\text{supprice} < \text{retprice}),$$

$$(\text{supprice} < 40), (\text{avlqty} > 0), (\text{state} = \text{FL}), (C_{(S,PS)}), (C_{(PS,P)})\}$$

Definition 13 The common selection conditions among all $n$ concurrent queries over a relation is denoted as $\Theta_I = \bigcap_{i=1}^{n} \Theta_i$. We also define $\Theta'_i$ as the set of selection conditions for a query $q_i$ (over a relation) that is not found in $\Theta_I$, i.e., $\Theta'_i = \Theta_i - \Theta_I$.

Definition 14 We can write an expression $\theta_I$ with all members of $\Theta_I$ in a conjunctive form, i.e., $\theta_I = \bigwedge_{\text{all}(\Theta_I)}$. In the same way, we define an expression $\theta'_i$ as $\bigwedge_{\text{all}(\Theta'_i)}$. Furthermore, for all the $n$ queries, we define an expression $\theta_D$ to connect all $\theta'_i$ expressions in a disjunctive form, i.e., we have $\theta_D = \bigvee_{i=1}^{n} \theta'_i$. 

25
4.2 Rule 1: Sequence of Selections

In traditional DBMSs, we have

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

(4.1)

In other words, a selection over a relation can be broken down to conjunctive conditions. Furthermore, we have

\[
\sigma_{\theta_1 \land \theta_2}(r) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(r)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(r))
\]  

(4.2)

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

\[
\sigma_{\theta_1}(r)\psi\sigma_{\theta_2}(r)\psi\ldots\psi\sigma_{\theta_n}(r) \neq \sigma_{\theta_1}(\sigma_{\theta_2}(\ldots(\sigma_{\theta_n}(r))\ldots)),
\]

and the above is true for any ordering of \(\sigma_{\theta_j}\) 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

$$
\sigma_{\theta_1}(r) \psi \sigma_{\theta_2}(r) \psi \ldots \psi \sigma_{\theta_n}(r) = \sigma_{\theta_1 \lor \theta_2 \ldots \lor \theta_n}(r)
$$

(4.3)

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 $R_1, R_2, \ldots, R_n$, we can recover each individual operand via

$$
\sigma_{\theta_i}(r) = f_i(P) = \Pi_{R_i} \sigma_{\theta_i}(P), \quad \forall i
$$

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 $f'_i(P') = \sigma_{\theta_i}(P')$ to recover all operands of the LHS as follows.

$$
\sigma_{\theta_i}[\sigma_{\theta_1 \lor \theta_2 \ldots \lor \theta_n}(r)]
$$

$$
= \sigma_{\theta_i}[\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) \cup \ldots \cup \sigma_{\theta_n}(r)]
$$

$$
= \sigma_{\theta_i}(r), \quad \forall i
$$

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

$$
\theta_i = \land_{all}(\Theta_i) = (\land_{all}(\Theta'_i) \land (\land_{all}(\Theta_I)) = \theta'_I \land \theta_I
$$

(4.4)

Therefore, the RHS of Eq. (4.3) becomes $\sigma_{\theta_0 \land \theta_i}(r)$, 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. (4.3).

**Corollary 2** The selection condition sets $\theta_D$ and $\theta_I$ can be applied sequentially, and in any order.

$$\sigma_{\theta_I \wedge \theta_D}(r) = \sigma_{\theta_D}(\sigma_{\theta_I}(r)) = \sigma_{\theta_I}(\sigma_{\theta_D}(r))$$

**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

$$\theta_I = \{(\text{supprice} < \text{retprice}) \wedge C_{(P,PS)} \wedge C_{(PS,S)}\}$$

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

$$\theta_D = \{(\text{supprice} > 40) \vee (\text{avlqty} > 0) \vee (\text{state} = FL)\}$$

*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. (4.3) 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 $\theta_I$. Corollary 1 says that reordering the (conjunctive) terms in $\theta_I$ 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.
4.3 Rule 2: Sequence of Projections

In traditional DBMSs, we have

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

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

\[ \Pi_{L_1}(r) \psi \Pi_{L_2}(r) \psi ... \psi \Pi_{L_n}(r) \cong \Pi_{\cup_{i=1}^n L_i}(r) \]  

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:

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

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 4.6), 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.

\[ q'_1: \text{SELECT pkey, mfgr, retprice FROM part P WHERE mfgr=mf2} \]

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

4.4 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.,

\[
\sigma_{\theta_1 \land \theta_2} (T_1 \bowtie_c T_2) \equiv (\sigma_{\theta_1} T_1) \bowtie_c (\sigma_{\theta_2} T_2)
\]  \hspace{1cm} (4.7)

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. (4.7), we have

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

**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. 3.1a): \( T_1 \omega_2 \), \( T_1 T_2 \), 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. \( \square \)

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

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

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 4.8.2.

Example 5 In the example database, if $q = \{q_2, q_3, q_4\}$ and $\Theta = \{ (\text{supprice} < \text{retprice}),
\text{(supprice} < 40), \text{(avqty} > 0), \text{(state} = \text{FL}), \text{C}_{(P,PS)}, \text{C}_{(PS,S)} \}$, the selection can be distributed as

$$\sigma_{(\text{rprice} > \text{sprice}) \lor (\text{supprice} < 40) \lor (\text{avqty} > 0) \lor (\text{state} = \text{FL})} (P \psi C_{P,PS} S P \psi C_{PS,S} S) =
\sigma_{(\text{rprice} > \text{sprice}) \lor (\text{supprice} < 40) \lor (\text{avqty} > 0) \lor (\text{state} = \text{FL})} (P \psi C_{P,PS} S) \psi C_{PS,S} \sigma_{(\text{state} = \text{FL})} S$$

4.5 Rule 4: Distributive Projection

In traditional DBMSs, a projection is *distributive* over joins.

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

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

\[
\Pi_{\cup_{i=1}^n L_i} (T_1 \psi T_2 \psi ... \psi T_n) \equiv \Pi_{L_1} (T_1) \psi \Pi_{L_2} (T_2) \psi ... \psi \Pi_{L_n} (T_n) \tag{4.8}
\]

**Proof:** Firstly, both sides generate a relation with the exact same schema \( \cup_{i=1}^n L_i \). 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. \( \square \)

Eq. (4.8) 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. (4.8), 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. (4.8) 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. (4.16) of Section 4.8.2. This rule was never mentioned in SharedDB and DataPath.
4.6 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., $q_i = \sigma_{\theta_i}(\times_{T \in T_i} T)$. 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., $T = \cup_i T_i$, and $C$ be the disjunctive form of all the conditions applied in individual queries, i.e., $C = \theta_1 \lor \theta_2 \lor \cdots \lor \theta_n$, we have

$$q_1 \psi q_2 \cdots \psi q_n \subseteq \psi C \forall T_j \in T \quad (4.9)$$

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 $Z = \times_{T \in T} T$, 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

$$LHS = \sigma_{\theta_1} Z \psi \sigma_{\theta_2} Z \cdots \sigma_{\theta_n} Z \supseteq \sigma C Z = \sigma C Z = RHS$$

If we consider the full outerjoin as $\psi$, the resulting table of RHS, denoted as $\mathcal{P}$, 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:

\[
f_i(\mathcal{P}) = \sigma_{\theta_i}(\mathcal{P}), \quad \forall i, 1 \leq i \leq n
\]

Note that the function \( f_i \) will return no tuples by working on subset (ii) of \( \mathcal{P} \), 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 \( \sigma_{\theta_i}(\sigma_{\mathcal{C}}\mathcal{Z}) = \sigma_{\theta_i}\mathcal{Z} \).

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

\[
q'_j = q_j \times \left( \bigotimes_{T \in \mathcal{T} - \mathcal{T}_j} \omega_T \right)
\]

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 \( \mathcal{Z} \). With this transformation, we follow the same reasoning in Case 1 to show that Eq. (4.9) 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. (4.9) 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. (4.9), 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 \( \mathcal{T} \), 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 \( \mathcal{T} - \mathcal{T}_j \)). To reduce that cost, we need to use the anti-join condition mentioned in Definition 9 to disallow such joined tuples.

With that, for aforementioned query \( q_j \), we replace its join condition with \( \theta_j \land \overline{\theta} \). 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. (4.9), we replace the RHS join condition \( C = \bigvee_{i=1}^{m} \theta_i \) with \( C = (\bigvee_i \theta_i) \lor (\bigvee_j (\theta_j \land \overline{\theta})) \). It is easy to see that the existence of \( \overline{\theta} \) conditions does not change the correctness of Eq. (4.9) – 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

\[
C = \{\theta_1 \lor \theta_2\} = \{(\text{retprice} < 40 \land \overline{\theta}) \lor (\text{supprice} < \text{retprice} \land \\
\text{supprice} < 40 \land C_{(P,PS)} \land C_{(PS,S)})\}
\]

By applying Rule 5, we get an RHS of \( P \psi_{C_1} \psi_{\overline{\theta}} \), by distributing the conditions in \( C \) to the relevant tables. Specifically, \( C_1 = \{(\text{retprice} < 40 \land \overline{\theta}) \lor (\text{supprice} < \text{retprice} \land \text{supprice} < \\
40 \land C_{(P,PS)})\} \) and \( C_2 = \{C_{(PS,S)}\} \).
4.7 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 15 A Group-By operation over a table $r$ with the grouping criterion $B$ (a subset of attributes of $r$), denoted as $B\mathcal{G}(r)$, returns an reorganized copy of $r$, in which tuples of $r$ form different groups based on the distinct values of $B$.

Definition 16 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 $B\mathcal{G}_{F(A_i)}(r)$, 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., $B\mathcal{G}_{F_1(A_1),F_2(A_2),...,F_k(A_k)}(r)$.

Note that Definition 16 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

$$B\mathcal{G}(r) \psi r \equiv B\mathcal{G}(r) \quad (4.10)$$

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

$$B_1 \mathcal{G}(r) \psi B_2 \mathcal{G}(r) \equiv B_2 \mathcal{G}(r) \quad (4.11)$$

**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

$$B_1 \mathcal{G}_{F(A_i), F'(A_i)}(r) \psi B_2 \mathcal{G}_{F(A_i), F'(A_i)}(r) \equiv B_2 \mathcal{G}_{F(A_i), F'(A_i)}(r) \quad (4.12)$$

**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

$$B_1 \mathcal{G}(r) \equiv B_2 \mathcal{G}(r) \quad (4.13)$$

**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

$$b_1 \mathcal{G}_{F(A_i)}(r) \sim b_2 \mathcal{G}_{F(A_j)}(r)$$

(4.14)

**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 $b_2 \mathcal{G}_{F(A_j)}(r) \sim b_1 \mathcal{G}_{F(A_i)}(r)$. □

**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 [37], 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-Bys. We believe this cost cannot be systematically eliminated, no matter we use CDBMS or traditional query-at-a-time systems.

### 4.8 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.

With the development of the PsiDB system model (Section 3.1) and equivalence rules (Section 4.1), we now discuss query optimization in PsiDB. We focus on algebraic optimization techniques while briefly comment on optimization at the physical plan level. Our vision is that query optimization in CDBMS should be considered at two levels. The first level falls into the paradigm
of traditional query optimization in which multiple execution plans of a query are evaluated. For CDBMS, we focus on a global query $Q$ built out of the individual queries. As a first step, we need to generate an algebraic expression as a representation of $Q$ that carries a lower cost than alternatives (e.g., so called algebraic optimization [38]). Then we can use the existing query optimizer to select a physical plan to compute $Q$. At the second level, we face the challenge of placing the queries in a given workload into different batches. Each batch consists of a number of queries with similar data access paths, and requires a different global query $Q$ to be chosen (as the first level challenge). Note that the second level challenge is unique in a CDBMS, and has never been studied in previous work.

4.8.1 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., $q_i = \Pi_{L_i} \sigma_{\theta_i}(\times_{T \in T_i} T)$, the global query can be rewritten by starting from Rule 5 and then applying Rule 2. The expression is

$$Q = \Pi_{(\bigcup_{i=1}^{n} L_i) \cup L_c} (\exists_{\exists_{T_j \in T}} T_j)$$

(4.15)

where $L_i$ is the set of attributes requested by query $q_i$ in its resulting table, and $L_c$ is the set of attributes involved in condition set $C$. Note that the RHS of Eq. (4.15) is basically that of Eq. (4.9) 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. (4.15) 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. (4.15) 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 $\theta_{i,j}$ with respect to the two involved tables $T_i$ and $T_j$. Then we just add $\theta_{i,j}$ 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. (4.15) is still valid upon adding another copy of $T_i$ to the set $\mathcal{T}$ and adding the self-join condition to $C$ as a disjunctive item.

Using equivalence rules presented in Chapter 4, we can create multiple execution plans for each batch. A brutal force search can be used to find all possible plans because having weak equality gives one way approach and limits the number of possible plans. An algorithm (Algorithm 1) is implemented to create those plans using the Rules in previous section. The function $P$ in the Algorithm 1 is a prediction function for processing time of PsiDB. We explain this function in section 5.3.1 along with batch-partitioning.

Algorithm 1 chooses to apply the equivalence rules based on their impact on a query batch. An example of this process is shown in Figure 4.1 which takes a batch of queries and generate a global query of $Q$. The general approach in the query optimizer is to predict if applying selections and projections on the more granular data result in lower processing time.

Figure 4.1 shows an example of applying rules to a batch of queries. The function $P$ is used to predict the processing time of each rule. In Fig. 4.1 $P(\text{rule}) >$ or $P(\text{rule}) <$ indicates if applying that rule cause lower processing time or higher.
4.8.2 Further Optimization 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 $T_i$ first, i.e., we get

$$Q = \Pi_{(\bigcup_{i=1}^{n} L_i) \cup \bigcup_{\forall T_j \in T} A}(\Delta C \bowtie \Pi_A(T_j))$$

(4.16)

(2) Eq. (4.15) 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 $\theta_I$ among all $\theta'_i$ (Definition 14) such that $C = (\theta'_1 \lor \cdots \lor \theta'_n) \land \theta_I = \theta_D \land \theta_I$, we could follow Rule 1 (Corollary 2) to reorder the selections.

(4) Following the discussions in (3), if the common conditions among all queries $\theta_I$ can be further written as $\theta_I = \theta_{T_1} \land \theta_{T_2} \cdots \theta_{T_m}$ 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 $\theta_{T_i}$.

---

In this figure "P(rule) > " or "P(rule) < " means if applying that rule cause lower processing time or higher compare to before using the rule.
Algorithm 1: Search Algorithm for PsiDB Query Optimization

Require: $q \leftarrow \{q_1, q_2, \ldots, q_n\}$
Ensure: $Q$

$Q \leftarrow \sigma_{\theta_1}(r)\psi \sigma_{\theta_2}(r)\psi \cdots \psi \sigma_{\theta_n}(r)$
if $P(\sigma_{\theta_1}(r)\psi \sigma_{\theta_2}(r)\psi \cdots \psi \sigma_{\theta_n}(r)) \leq P(\sigma_{\theta_1} \vee \theta_2 \ldots \vee \theta_n(r))$ then
$Q \leftarrow \sigma_{\theta_1}(r)\psi \sigma_{\theta_2}(r)\psi \cdots \psi \sigma_{\theta_n}(r)$
if $P(\Pi_L Q) \geq P(\Pi_{L_1}(\sigma_{\theta_1}(T_1))\psi \Pi_{L_2}(\sigma_{\theta_2}(T_2))\psi \cdots \psi \Pi_{L_m}(\sigma_{\theta_n}(T_m)))$ then
$Q \leftarrow \Pi_{L_1}(\sigma_{\theta_1}(T_1))\psi \Pi_{L_2}(\sigma_{\theta_2}(T_2))\psi \cdots \psi \Pi_{L_m}(\sigma_{\theta_n}(T_m))$
else
$Q \leftarrow \Pi_L Q$
end if
else
$Q \leftarrow \sigma_{\theta_1} \vee \theta_2 \ldots \vee \theta_n(r)$
if $P(Q) \geq P(\sigma_{\theta_1}(T_1)\psi (\sigma_{\theta_2} T_2)\psi \cdots \psi (\sigma_{\theta_n} T_m))$ then
$Q \leftarrow (\sigma_{\theta_1}(T_1)\psi (\sigma_{\theta_2} T_2)\psi \cdots \psi (\sigma_{\theta_n} T_m)$
if $P(\Pi_L Q) \geq P(\Pi_{L_1}(\sigma_{\theta_1}(T_1))\psi \Pi_{L_2}(\sigma_{\theta_2}(T_2))\psi \cdots \psi \Pi_{L_m}(\sigma_{\theta_n}(T_m)))$ then
$Q \leftarrow \Pi_{L_1}(\sigma_{\theta_1}(T_1))\psi \Pi_{L_2}(\sigma_{\theta_2}(T_2))\psi \cdots \psi \Pi_{L_m}(\sigma_{\theta_n}(T_m))$
else
$Q \leftarrow \Pi_L Q$
end if
else
$Q \leftarrow \Pi_{L_1}(\sigma_{\theta_1} \vee \theta_2 \ldots \vee \theta_n(r))$
end if
end if
end if

4.8.3 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. (4.15) or Eq. (4.16). Note that we have $C = \theta_1 \vee \theta_2 \vee \cdots \vee \theta_n$, therefore $q_i$ can be obtained by

$$q_i = \Pi_{L_i} \sigma_{\theta_i}(P)$$ (4.17)

where $P$ is the global query result. Another fact of interest is that the recovering function is almost identical to the query expression itself (recall $q_i = \Pi_{L_i} \sigma_{\theta_i}(\bigwedge_{T \in T_i} T)$). 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 $L_c$ in Eq. (4.15) – as each tuple arrives, we also need to apply the selection with predicates involving attributes in $\theta_i$ but not in $L_i$.

Eq. (4.17) shows, due to the streaming nature of selection and projection, each tuple can be processed on-the-fly and discarded afterwards.  

4.9 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.

4.9.1 Cost of Computing $P$

In Eq. (4.15), 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 selective therefore index-based join algorithms can be utilized. This can be addressed in the following aspects.

First, for a multitude of queries in the workload, the combined number of tuples accessed in the table can dwarf the whole table therefore it is efficient to conduct a table scan (by only considering the total number of tuples accessed). Such “sharing” of computation is the main driving force behind CDBMSs, and the SharedDB paper [7] reported an elegant quantitative analysis on it.

Second, recent database system research revealed even greater performance advantage of table scans, and that builds the foundation for the success of PsiDB. Due to better utilization of cache,

---

2If duplicate elimination is required, two step processing is needed – same as ‘Group By’ discussed in Section 4.7.
memory, and disk resources (e.g., sequential I/O), a table scan is believed [39] to be more efficient than (secondary) index access even under low selectivity (i.e., the percentage of data returned). A more systematic study [28] confirmed such findings – a break-even point (in selectivity) of less than 1% for an index-based scan to outperform table scan is reported for various in-memory systems. This means the indexed scan wins only when it accesses less than 1% of the data in a table, and this is highly unlikely in a PsiDB workload. For disk-based databases, the advantage of scanning is even bigger (the break-even point is less than 0.1%).

Figure 4.2 is created using TPC-H tables and joining them together. To create this table, the same queries with the same selectivities run by enforcing Scan and Index as access path to data. The ration of those queries are reported in Figure 4.2. The tables and their joins are shown in Table 4.1. Using SQL-Server as the database, enforcing the access path to the tables requires the option parameter of FORCESCAN or FORCEINDEX. The following a query example is shown with access path option.

```
SELECT *
FROM PART AS p INNER JOIN PARTSUPP AS ps ON p.part_id = ps.part_id
WITH (FORCESCAN)
WHERE ps.availqty <= x;
```

To verify the above results, we ran a series of queries by forcing the optimizer to choose either index or table scan. Fig. 4.2 shows the joins between tables in TPCH database. Each bar represent the join among one (no-join only accessing the tuples) to more tables. According to Fig. 4.2, the break-even point is at 1.11% selectivity for a in-memory database (SF10) and 1% for disk-based

<table>
<thead>
<tr>
<th></th>
<th>Parts</th>
<th>P</th>
<th>Partsupp</th>
<th>PS</th>
<th>Suppliers</th>
<th>S</th>
<th>Lineitem</th>
<th>L</th>
<th>Orders</th>
<th>O</th>
<th>Customers</th>
<th>C</th>
<th>Nations</th>
<th>N</th>
<th>Regions</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>P</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>O</td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>Partsupp</td>
<td>PS</td>
<td></td>
<td>PS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>Suppliers</td>
<td>S</td>
<td></td>
<td>PS</td>
<td></td>
<td>S</td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>Lineitem</td>
<td>L</td>
<td></td>
<td>PS</td>
<td></td>
<td>S</td>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>Customers</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>Regions</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PS</td>
<td>S</td>
<td>L</td>
<td></td>
<td>O</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 4.1: The tables and their relations of Figure 4.2
database (SF100). The same trends can be observed for single table search as well as multi-table joins.

![Graph showing speedup of three queries via sequential scan over index-based access](image)

Figure 4.2: Speedup of three queries via sequential scan over that by index-based access.

4.9.2 Size of $P$

In addition to the discussion in 3.3.1 about the size of the resulting relation, another concern is the size of $P$ 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 $P$ will appear in the resulting table of at least some query $q_i$.

**Proof:** This can be shown via Eq. (4.15): if a tuple passed the condition $C$ from the intermediate table resulted from $\bowtie c T_j$, 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

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

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 6.1.2.3).

4.9.3 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. (4.17)). 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.
4.10 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 4.8.2, 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. (4.15) or Eq. (4.16). 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 [40], and design of the query result distributor mentioned in Section 4.8.3.
Chapter 5: Batch Partitioning

5.1 A Research Gap

As mentioned above, query optimizer plays a key role in any database system. Query optimizer techniques have been an active research topic in the last four decades. Up to this day, there are three main lines/streams of research:

1. Given a single query, how to efficiently process that query and retrieve the results?

   In the traditional database systems, each query is excepted to be executed independently [41, 42, 43, 44]. Figure 1.1 shows all the components of an RDBMS for executing a single query. A request from the user translates to a query, and the query will go through the parser, optimizer and executer to get the data from storage and send the result back to the user [45, 46]. So, the underlying idea of this line of research is to compute optimal execution plans for a single query. As a result of this line of research, several query optimizer techniques are developed such as indexing, sequential scanning, etc [47, 48, 49, 28].

2. Given a set of queries, how to efficiently process all of them as a single batch?

   The landscape of data management systems has been changing dramatically in the last decade, and traditional RDBMSes are frequently criticized for not meeting requirements of modern data-intensive applications. Today’s database systems often need to serve a multitude of user queries in a short period of time. For example, in Alibaba, a leading Chinese online retailer, needs to process a workload at the rate of 10,000 query per second. Under such high demand, the one-query-at-a-time model falls short in meeting performance require-
ments because it can easily lead 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 processing multiple queries have been reported [32, 7, 6, 20, 5].

In light of the above, the database community developed several database systems that combines a set of queries into a single batch and attempts to efficiently process the batch. Examples include: Relational Database Management Systems for Batch Processing i.e. PsiDB and SharedDB designed for the workload with large number of users. These databases are capable of handling a set of queries instead of one single query. The idea of this line of research is demonstrated in Figure 5.1. The query optimizer in Figure 5.1 gets a set of queries $q_1, \ldots, q_n$ and combines them into one query $S$ so it can be processed by the database. After executing the combined query, $P$ is the table containing all information required by the set of queries, which in the next step is distributed among them.

3. Given a set of queries, how to efficiently partition them into batches and process each batch individually?

While there are many studies for the first and second lines of research, there are no studies for this line of research (to the best of our knowledge). Therefore, the goal of our research is to study whether partitioning the queries into batches can improve the batch-processing database systems performance.
The idea of such a database system is demonstrated in Figure 5.2. In this database system, a set of queries $q_1, \ldots, q_n$ are partitioned into multiple batches $S_1, \ldots, S_v$. Associated with each batch $S_i$, there is a result $P_i$, leading into a set of results: $P_1, \ldots, P_v$. Each $P_i$ is then distributed among the queries in batch $i$.

Figure 5.2: A batch-processing model with partitioned query. Here $t_1, t_2, \ldots, t_m$ are Tables, $q_1, \ldots, q_n$ are the queries, $S_1, S_2, \ldots, S_v$ are batches of queries and $P_1, P_2, \ldots, P_v$ are output data for each batch.

### 5.2 Problem Description

Batching is one of the main functions of modern databases due to how directly it affects the performance of data retrieval. On a PsiDB, the performance is impacted by the way we batch a set of queries because it impacts the required run time to retrieve the data. In this research, the main goal of the query batching problem (QBP) is to find the best way of partitioning a given set of queries into batches in order to minimize the total processing time. For example, let $Q = \{q_1, q_2, q_3\}$ be a set of queries that needs to be processed. In this case, $Q$ can be processed in 5 different ways by partitioning it into 1, 2, or 3 batches as follows: $\{(q_1), (q_2), (q_3)\}$, $\{(q_1, q_2), (q_3)\}$, $\{(q_1, q_3), (q_2)\}$, $\{(q_1), (q_2, q_3)\}$, and $\{(q_1, q_2, q_3)\}$. The total processing time of each solution of the query batching problem may vary due to the correlation between the queries. The correlation between queries can be established by comparing the data that each query is attempting to retrieve. For example, if $q_1,$
$q_2$, and $q_3$ request similar data from similar tables, it will probably be faster to process them as a single batch.

Overall, in order to solve the query batching problem, it is important to know how long the processing time of any potential batch would be. However, computing such a processing time is not a trivial task. This is because to process a batch, a database system typically takes two steps and each one requires some computational efforts [32, 4]. In the first step, it combines/converts all the queries in the batch into a single global query containing all the attributes and records of each query in the batch. The database system will then process the global query and retrieve the entire data requested by the global query. It is also worth mentioning that processing the global query in an efficient manner is not a trivial task by itself because it requires operations such as joining tables and filtering. More importantly, the amount of data retrieved as a result of processing the global query is unknown in advance. In the second step, some additional computational efforts should be made for searching the retrieved data (that its size is unknown in advance) and distributing it among queries in the batch. Due to the existence of unknown parameters in these two steps, computing the exact total processing time of a given batch (in advance) is a challenging task that depends highly on the single-batch-processing method used.

In light of the above, if one can generate a function for predicting the processing time of any possible batch, then a mathematical program for obtaining an optimal solution of the QBP can be developed. Specifically, let $Q := \{q_1, \ldots, q_n\}$ be the set of queries. Observe that since the total number of queries is $n$, the maximum number of possible batches would be $n$, i.e., each batch will contain exactly one query. We denote batch $j \in \{1, \ldots, n\}$ by $S_j \subseteq Q$ and its predicted processing time by $P(S_j)$. By using these notations, the QBP can be stated as follows:

$$\min_{S_1, \ldots, S_n \subseteq Q} \left\{ \sum_{j=1}^{n} P(S_j) : \bigcup_{j=1}^{n} S_j = Q \text{ and } S_i \cap S_j = \emptyset \forall i, j \in \{1, \ldots, n\}, i \neq j \right\}.$$
Note that due to the significant variability of processing times in practice, it would be hard to predict the processing times accurately. One heuristic way to reduce the variability of data is to employ log$_2$-transformation. So, in the remaining of this paper, $P(S_j)$ basically represents the prediction of the log$_2$-transformation of the processing time for batch $S_j$. In other words, whenever we say the ‘predicted processing time’, we mean the ‘predicted log$_2$-transformation of processing time’.

Finally, it is worth mentioning that the main/natural assumption of this study is that queries are not redundant. Specifically, we assume that each query in $Q$ should involve (i.e., search) at least one of the tables in the database system under consideration.

5.3 The Proposed Approach

In this section, we explain our proposed approach for solving the query batching optimization problem. Our approach contains three phases as follows.

1. Developing the batch processing time prediction function

2. Developing an optimization model to effectively partition the queries into batches

3. Developing effective solution methods to solve the optimization model

In the remaining of this section, we will provide further details about each phase.

5.3.1 The Batch Processing Time Prediction Function

In this section, we develop a function $P(S')$ that serves to predict batch processing time for any arbitrary query batch $S'$ of a given database. An ideal function $P(S')$ has the two following desired characteristics: (1) The function has high accuracy; and (2) The function is simple as the complexity of the optimization model for the QBP and its solution approaches highly depend on it. The use of the QBP is justified only if the sum of the solution time of the QBP and the processing
time of the batches prescribed by the QBP is dominated by the processing time of the same set of queries when the QBP is not employed. Accordingly, if one cannot solve the QBP quickly then the benefit of batching may not be significant.

As mentioned in Section 5.2, for a given query batch $S'$, its predicted (log$_2$-transformation of) processing time, $P(S')$, depends on the size of the data that must be retrieved for processing batch $S'$. However, such information is not known in advance and becomes available only while executing the batch. Therefore, in our approach we benefit from some other attributes (or features) of $S'$ that are either available in advance or easy to compute to predict the processing time of batch $S'$.

Let $T := \{1, \ldots, m\}$ denote the index set of all tables in the database (where $m$ is the total number of tables). To predict the processing time of a batch $S'$, we introduce $|T^{S'}| + 1$ attributes associated with batch $S'$, where $T^{S'} \subseteq T$ denotes the index set of all tables required by the queries in batch $S'$. The first $|T^{S'}|$ attributes are indicators of the size of tables required for batch $S'$. Specifically, we introduce the attribute log$_2(s_t)$ for each $t \in T^{S'}$ with $s_t$ being the number of records in table $t$. Note that the size of tables in a database can be significantly different and hence the proposed log$_2$-transformation can be helpful in normalizing the size of the tables in such cases. We also introduce one additional attribute, indicating the number of queries in batch $S'$, denoted by $n'$. Using the proposed attributes, we construct a function that can predict the (log$_2$-transformation of) processing time of any given batch. This function takes the following form.

$$P(S') := [\beta_0 + \beta_1 n' + \sum_{t \in T} \beta_{t+1} \log_2(s_t) y'_t]^2,$$  \hspace{1cm} (5.1)

where $y'_t \in \{0, 1\}$ is a parameter indicating whether table $t$ is required to process queries in batch $S'$. Also, $\beta_0, \beta_1, \ldots, \beta_{|T|+1} \in \mathbb{R}$ are the coefficients that should be estimated. In Section 6.2, we will numerically show that the proposed function has high prediction accuracy on the two different database benchmarks that we consider. This implies that the proposed function addresses the
first characteristic required, mentioned at the beginning of this section. Additionally, we observe that the proposed \( P(S') \) is a quadratic function of \( n' \) and \( y'_1, \ldots, y'_{|T|} \). Despite being a quadratic function, \( P(S') \) can still be handled easily by commercial solvers such as CPLEX and Gurobi when used in the optimization model that we develop for the QBP, which satisfies the second desired characteristic mentioned above. Note that we do not use a linear function of the form,

\[
\beta_0 + \beta_1 n' + \sum_{i \in T} \beta_{i+1} \log_2(s_i)y'_t
\]

instead of Eq. (A.1), since Eq. (5.2) has a significant weakness when being employed in the QBP. In Section 5.3.2, we will explain in detail why such a linear function is not suitable.

We estimate coefficients \( \beta_0, \beta_1, \ldots, \beta_{|T|} \) through a training procedure over a large set of randomly generated batches of queries. In Section 6.2, we discuss in details how a random batch is generated. For each batch \( S' \), one can compute the value of each attribute and then actually process the batch to compute its real processing time, denoted by RealTime. The goal is to estimate the coefficients of the proposed processing time prediction function \( P(S') \) such that,

\[
P(S') \approx \log_2(\text{RealTime}).
\]

This implies that,

\[
\sqrt{P(S')} = \beta_0 + \beta_1 n' + \sum_{i \in T} \beta_{i+1} \log_2(s_i)y'_t \approx \sqrt{\log_2(\text{RealTime})}.
\]

This itself implies that in order to estimate the coefficients of the proposed function, one can simply fit a linear regression function. Specifically, \( \sqrt{\log_2(\text{RealTime})} \) can serve as the dependent variable and the proposed attributes can serve as independent variables. In this study, we employed the package `linear_model` from `scikit-learn` in Python to compute the coefficients based on Ridge linear regression [50].
5.3.2 The Optimization Model

In this section, we develop a MBQP to solve the QBP using the proposed processing time prediction function. In the remaining of this section, it is assumed that coefficients $\beta_0, \beta_1, \ldots, \beta_{|T|+1}$ are already estimated and are available as parameters. However, the assignment of the queries to batches, and consequently the number of queries in each batch and the set of tables required for processing each batch are unknown and are to be determined by the optimization model. Let $Q_t \subseteq \{1, \ldots, n\}$ be the set of queries that requires table $t \in T$. Note that, by assumptions, each query requires at least one table. Given that the maximum number of possible batches built out of a set of $n$ queries is $n$, let $x_{ij}$ be a binary decision variable that takes the value of one if query $i \in \{1, \ldots, n\}$ is assigned to batch $j \in \{1, \ldots, n\}$, and zero otherwise. Also, let $y_{jt}$ be a binary decision variable indicating whether table $t \in T$ is used by at least one query in batch $j \in \{1, \ldots, n\}$. For each batch $j \in \{1, \ldots, n\}$, we introduce the continuous variable $n_j$ for indicating the number of queries in batch $j$. As an aside, although $n_j$ is defined as continuous decision variable it will naturally take integer values in the proposed formulation for each batch $j \in \{1, \ldots, n\}$. Finally, for each batch $j \in \{1, \ldots, n\}$, we introduce the binary decision variable $z_j$ which equals 1 at least one query is assigned to batch $j$. Using this notation, the query batching problem can be formulated as the following MBQP,

$$\begin{align*}
\min & \sum_{j=1}^{n} \left[ \beta_0 z_j + \beta_1 n_j + \sum_{t \in T} \beta_{t+1} \log_2(s_t)y_{jt} \right]^2 \\
\text{s.t.} & \sum_{j=1}^{n} x_{ij} = 1 & \forall i \in \{1, 2, \ldots, n\} \\
& n_j = \sum_{i=1}^{n} x_{ij} & \forall j \in \{1, 2, \ldots, n\} \\
& \sum_{i \in Q_t} x_{ij} \leq n y_{jt} & \forall j \in \{1, 2, \ldots, n\}, \forall t \in T \\
& \sum_{i \in T} y_{jt} \leq m z_j & \forall j \in \{1, 2, \ldots, n\}
\end{align*}$$ (5.3)
where the objective function (5.3) measures the total predicted processing time of non-empty batches. Constraint (5.4) ensures that each query is assigned to exactly one batch. Constraints (5.5)-(5.7) ensure that for each batch \( j \in \{1, \ldots, n\} \) and table \( t \in T \), decision variables \( n_j, y_{jt}, \) and \( z_j \) follow their definitions, respectively. Specifically, Constraint (5.5) guarantees that the value of \( n_j \) is computed properly for any batch \( j \in \{1, \ldots, n\} \). Constraint (5.6) ensures that if table \( t \in T \) is required for the set of queries in batch \( j \in \{1, \ldots, n\} \) then \( y_{jt} = 1 \). Finally, Constraint (5.7) guarantees that if batch \( j \) needs some tables to be processed, i.e., \( y_{jt} = 1 \) for some \( t \in T \), then \( z_j = 1 \) (because the batch \( j \) must be non-empty in that case).

Observe that the objective function of the proposed formulation simply captures \( \sum_{j=1}^{n} P(S_j) \) where \( P(S_j) \) is the quadratic function obtained in Section 5.3.1. So, it is natural to ask why cannot \( P(S_j) \) be a linear function, i.e., Eq. (5.2)? Note that the objective of the QBP is to minimize the total processing time of a set of queries by regrouping them into batches. A linear function may have high accuracy for predicting the processing time of each batch individually but when used in the optimization framework for minimizing the total processing time, it will perform poorly. Specifically, if one uses a linear processing time prediction function, the objective function of the proposed formulation will change to,

\[
\sum_{j=1}^{n} \beta_0 z_j + \beta_1 n_j + \sum_{t \in T} \beta_{t+1} \log_2(s_t)y_{jt} = \beta_1 n + \sum_{j=1}^{n} \beta_0 z_j + \sum_{t \in T} \beta_{t+1} \log_2(s_t)y_{jt}.
\]

This implies that the number of queries in the batches will not play any role in the optimization process because \( \beta_1 n \) is a constant. In that case, by assuming that \( \beta_0, \beta_1, \ldots, \beta_{|T|} \geq 0 \) (which is likely to be the case in practice), the optimal solution for the QBP is to create only one batch containing all queries. Therefore, such linear functions are not suitable for showing the effectiveness of batching. Hence, a main advantage of using the proposed quadratic function is that it allows to capture the
importance of the number of queries assigned to different batches. We next present a series of techniques to strengthen the proposed formulation.

One simple technique is to dis-aggregate Constraints (5.6)-(5.7) and replace them by the following constraints to remove fractional solutions [51]

\[ x_{ij} \leq y_{jt} \quad \forall i \in Q_t, \forall j \in \{1, 2, \ldots, n\}, \forall t \in T \]  \hspace{1cm} (5.9)

\[ y_{jt} \leq z_j \quad \forall t \in T, \forall j \in \{1, 2, \ldots, n\}. \]  \hspace{1cm} (5.10)

Also, the proposed formulation is suffering from symmetry issues because different representation of a same solution exist. For example, suppose that \( n = 3 \) and consider the solution that process all queries together (as a single batch). Obviously, because \( n = 3 \), the formulation creates three potential batches and so three possible representations for the solution exist,

- **Representation I**: Batch 1 = \{1, 2, 3\}, Batch 2 = \{\}, Batch 3 = \{\};

- **Representation II**: Batch 1 = \{\}, Batch 2 = \{1, 2, 3\}, Batch 3 = \{\};

- **Representation III**: Batch 1 = \{\}, Batch 2 = \{\}, Batch 3 = \{1, 2, 3\}.

As a result of the existing symmetry, one can expect that commercial solvers such as CPLEX and Gurobi struggle to solve the proposed MBQP without applying symmetry breaking techniques [52]. To break the symmetry, we propose to add the following constraints to the model, i.e.,

\[ x_{ij} = 0 \quad \forall i, j \in \{1, \ldots, n\} \text{ and } i > j \]  \hspace{1cm} (5.11)

Observe that without using the proposed symmetry breaking constraints, a query \( i \in \{1, \ldots, n\} \) can be assigned to any batch \( j \in \{1, \ldots, n\} \) in the optimization model. This is in fact one of the main reasons that symmetry exists in our proposed formulation. Hence, our proposed symmetry breaking technique limits this flexibility by forcing the optimization model to assign query \( i \in \{1, \ldots, n\} \)
{1, \ldots, n} to only a batch \( j \in \{1, \ldots, n\} \) with \( j \leq i \). This implies that query 1 is only allowed to be assigned to batch 1, query 2 is only allowed to be assigned to either batch 1 or 2, query 3 is only allowed to be assigned to either batch 1 or 2 or 3, and so on. This process obviously does not remove any feasible solution of the QBP and will (partially) break the symmetry in the proposed formulation. As an aside, we observe that both Representation II and III, for the example given earlier, will become infeasible by using the proposed symmetry breaking technique. Finally, in terms of implementation, instead of explicitly adding the proposed symmetry breaking constraints, it would be computationally better not to generate \( x_{ij} \) for all \( i, j \in \{1, \ldots, n\} \) with \( j > i \) when creating the model. This efficient implementation is used in this study.

5.3.2.1 Computational Complexity

We now explore the computational complexity of the QBP when its objective function is the proposed quadratic function. However, before that, we make an important observation. That is if \( \beta_0, \beta_1, \ldots, \beta_{m+1} \geq 0 \) then the proposed MBQP can compute the objective value of any optimal solution accurately. However, for non-optimal (feasible) solutions, the proposed formulation may not capture their objective values accurately. Note that if \( t \in \{0, 1, \ldots, m+1\} \) such that \( \beta_t < 0 \) then the proposed formulation may not even capture the objective value of optimal solutions properly. In order to ensure that the proposed formulation can capture the objective value of any feasible solution accurately, the following additional inequalities should be added,

\[
y_{jt} \leq \sum_{i \in Q_t} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T \\
(5.12)
\]

\[
z_j \leq n_j \quad \forall j \in \{1, 2, \ldots, n\}. \\
(5.13)
\]

Constraint (5.12) ensures that if for any given batch \( j \in \{1, \ldots, n\} \) table \( t \in T \) is not needed, i.e., \( \sum_{i \in Q_t} x_{ij} = 0 \), then we must have that \( y_{jt} = 0 \). Finally, Constraint (5.13) guarantees that if no
query is assigned to a given batch \( j \in \{1, \ldots, n\} \), i.e., \( n_j = 0 \), then we must have that \( z_j = 0 \). These two additional inequalities combined with the assumption that each query should involve at least one table guarantee that the (predicted) processing time of each batch is computed properly. This implies that if a batch is empty and/or do not need to explore any table then its associated (predicted) processing time is zero. Also, if a batch does not need some of the tables then those will not impact the processing time of that batch.

Although Constraint (5.12) and (5.13) are not necessary for computing optimal solutions for the QBP when \( \beta_0, \beta_1, \ldots, \beta_{m+1} \geq 0 \), for the purpose of studying the computational complexity of the QBP we assume that they are included in the formulation. This is because for identifying the computational complexity of the QBP, the decision problem of the QBP should be explored. The decision problem of the QBP (when employing the proposed quadratic function), denoted by QBP, can be stated as follows: does there exist a solution that can satisfy the following set of constraints,

\[
\sum_{j=1}^{n} \left[ \beta_0 z_j + \beta_1 n_j + \sum_{t \in T} \beta_{t+1} \log_2(s_t) y_{jt} \right]^2 \leq U
\]

\[
y_{jt} \leq \sum_{i \in Q_t} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
z_j \leq n_j \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\}
\]

\[
n_j = \sum_{i=1}^{n} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
\sum_{i \in Q_t} x_{ij} \leq n y_{jt} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
\sum_{i \in T} y_{jt} \leq m z_j \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
n_j \geq 0, x_{ij}, y_{jt}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]
where $U$ is a given parameter. The QBP is basically a feasibility problem in which the objective function of the proposed MBQP has changed to a constraint. Therefore, in order to make an accurate analysis on the computational complexity of the QBP, the objective value of each feasible solution must be captured accurately in the QBP. That is the main reason that Constraints (5.12) and (5.13) are included in the QBP.

**Theorem 13** The query batching problem when employing the proposed processing time prediction function (A.1) is NP-hard.

**Proof.** The proof is rather lengthy and is provided in the appendix.

5.3.3 Heuristic Solution Methods

Observe that the size of the proposed formulation is $O(mn^3)$, because the number of constraints is $O(mn)$ and the number of variables is $O(n^2)$. This combined with Theorem 13 imply that there is little hope that the proposed MBQP can be solved quickly enough for practical-sized instances with possibly thousands of queries and multiple tables. In Section 6.2, we will show that instances with up to 500 queries on a database benchmark with 5 tables and around 10 GB of data can be solved to optimality using commercial solvers within a few hours (on a powerful computing node). Obviously, for larger instances, the solution time for the MBQP is expected to increase significantly. As a result, even if optimal or near optimal batching can offer some savings in terms of processing time of a set of queries, these savings can be dominated by long solution time of the MBQP. As a consequence, in this section, we propose two simple heuristic algorithms permitting us to quickly generate high-quality feasible solutions to the proposed MBQP.

We refer to the first algorithm as the Restricted-Cardinality Search Method I (RCSA-I). The underlying idea of RCSA-I is to reduce the size of the proposed formulation and its corresponding feasible solution space. To that end, RCSA-I imposes a bound, denoted by $k$ (where $k \leq n$), on the maximum number of batches allowed to be formed in the proposed formulation. To apply
this restriction, all instances of \( j \in \{1, \ldots, n\} \) in the MBQP should be replaced by \( j \in \{1, \ldots, k\} \), resulting into a restricted formulation as follows.

\[
\begin{align*}
\min & \quad \sum_{j=1}^{k} \left[ \beta_0 z_j + \beta_1 n_j + \sum_{t \in T} \beta_{t+1} \log_2(s_i)y_{jt} \right]^2 \\
\text{s.t.} & \quad \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\} \\
& \quad n_j = \sum_{i=1}^{n} x_{ij} \quad \forall j \in \{1, 2, \ldots, k\} \\
& \quad \sum_{i \in Q} x_{ij} \leq n y_{jt} \quad \forall j \in \{1, 2, \ldots, k\}, \forall t \in T \\
& \quad \sum_{t \in T} y_{jt} \leq m z_j \quad \forall j \in \{1, 2, \ldots, k\} \\
& \quad n_j \geq 0, x_{ij}, y_{jt}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, k\}, \forall t \in T.
\end{align*}
\] (5.14)

We refer to the new formulation (5.14)-(5.19) as the MBQP-I. Observe that for small values of \( k \), the size of MBQP-I is significantly smaller than the (original) MBQP. Hence, one can expect to solve the MBQP-I faster using commercial solvers. To improve the solution time even further, RCSA-I restricts the formulation even more by adding the following set of constraints to the MBQP-I,

\[
\sum_{j=1}^{k} y_{jt} \leq 1 \quad \forall t \in T.
\] (5.20)

Constraint (5.20) forces all queries requiring a given table \( t \) to be assigned to the same batch. The addition of Constraint (5.20) results in a restricted variant of MBQP-I that we refer to as the MBQP-II and it plays a key role in RSCA-I. Starting from \( k = 1 \), RCSA-I iteratively increases \( k \) with a stepsize one and solves the corresponding MBQP-II in each iteration. As an aside, we note that the solution for \( k = 1 \) is trivial and hence MBQP-II does not need to be solved for \( k = 1 \). If in a given iteration there is an improvement in the optimal objective value of MBQP-II compared to
the previous iteration, the algorithm increases $k$ by one and starts a new iteration. Otherwise, the algorithm terminates and returns the best solution found and the corresponding number of batches, $k^*$. It is worth mentioning that for implementing RCSA-I, three points should be considered. First, the solution of each iteration can be used as a warm-start for the next iteration. Second, all the proposed techniques (in Section 5.3) for strengthening the proposed MBQP can be used for strengthening MBQP-I and MBQP-II. Third, because of Constraint (5.20), the maximum number of iteration of RCSA-I is $m$.

We refer to the second algorithm as the Restricted-Cardinality Search Method II (RCSA-II). RCSA-II is a two-phase method and its underlying idea is to improve the solution obtained by RCSA-I further (if possible). Hence, RCSA-II first calls RCSA-I in its first phase. Let $k^*$ be the value of $k$ in the last iteration of RCSA-I. For the second phase, the algorithm initializes $k$ by $k^*$ and then iteratively solves the MBQP-I (instead of MBQP-II). Similar to the first phase, the algorithm increases the value of $k$ by one as long as some improvements in the optimal objective value of MBQP-I is observed (compared to its last iteration). Otherwise, the algorithm terminates and returns the best solution found. It is worth mentioning that the implementation points discussed for RCSA-I should also be considered for the RCSA-II.
Chapter 6: PsiDB Experimental Results

6.1 PsiDB Evaluation

We accomplished a Level I implementation (Section 4.10) 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 6.1.1.2) 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. (4.16). The query \( Q \) is then transformed into an SQL statement and sent back to SQL Server to compute the final table \( P \). We collect the tuples of \( P \) in batches, and each batch is sent to an in-memory data filtering module that applies Eq. (4.17) to generate the individual query results. In short, this is a simple PsiDB implementation without considering any physical level optimizations.

6.1.1 Experimental Setup

6.1.1.1 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
Table 6.1: Speedup of PsiDB over SQL Server under different database sizes and query numbers

<table>
<thead>
<tr>
<th>Coverage</th>
<th>SF1</th>
<th>SF10</th>
<th>SF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

**6.1.1.2 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 a 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 $P$).
As a result, our workloads all consist of a heterogeneous set of queries with different join skeletons (Table 6.1). Note that there are up to eight tables joined in the same query in TPC-H.

<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>

### 6.1.1.3 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 [53], 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 6.1.3.1.

### 6.1.2 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. (4.9)) 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*.

![Graphs showing time for processing Q and tuple distribution under SF100 for different data coverage percentages](image)

Figure 6.2: Time for processing $Q$ and tuple distribution under SF100

### 6.1.2.1 Overall Performance

Fig. 6.1 shows the speedup of PsiDB over SQL Server in processing the same workload. The performance of MySQL 8.0 is consistently (3.5-5 times) lower than SQL Server therefore we only report SQL Server results here. Exact running time of such experiments are shown in Appendix 6.1.3. In general, PsiDB outperforms SQL Server in all cases, with a speedup up to 36X. The speedup increases with the following factors: (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.

For factor (2), we want to point out that it is related to the size of $\mathcal{P}$, and costs of filtering tuples from $\mathcal{P}$ are found to be the dominating factor in the total workload processing time (more details in the next paragraph). 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 [54]. 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 \textit{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 4.9: when system is bound by disk I/O, the scan-based algorithm shows more advantage over indexed access.

6.1.2.2 Lower Bound

Owning to the fact that the system would never have a lower performance in compare to one-query-at-a-time for batches larger than 64 queries, it is important to define and compare the worst case scenario under different workloads and query batches. The lower bound of \( \Psi \text{DB} \) would happen when the selectivity of the queries in a batch leads to zero and 100% of the data has to load into the memory. In that case the performance drops down to 1.5x-3x as it is shown in Fig. 6.1. The main reason is the increase of computational load in distribution phase. Fig. 6.2 shows with increasing the coverage, the processing time of the distribution dominates the performance.

![Figure 6.3: \( \sum |p_i|/|P| \) with 1%, 50% and 100% for SF10](image)
Figure 6.4: Total resource consumption of PsiDB and SQL Server in running workloads SF1, SF10 and SF100 and 50% data coverage
Figure 6.5: Speedup of PsiDB over SQL Server with batches including queries with group by and aggregates
6.1.2.3 Cost of Computing $\mathcal{P}$

By looking at the break-down of the workload processing time for PsiDB (Fig. 6.2), 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 6.3 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.

6.1.2.4 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. 6.4 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.
6.1.3 More Experimental Results

In addition to shown results, we compared \( Q \) time versus Distribution time for all of our Scale Factors (Fig. 6.7c). Also we show all the comparison between SQL Server vs PsiDB for all Scale Factors and Selectivities discussed before in Figure 6.8.

6.1.3.1 (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. 6.6, 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).

6.1.3.2 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. 6.5 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. 6.1), 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. 6.1, which again shows the impact of computation on PsiDB performance.
6.1.3.3 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. 6.2. 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 $\mathcal{P}$ 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 $\mathcal{P}$). 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.

6.2 A Computational Study on Partitioning Batches

In this section, we investigate the performance of our proposed approach on two database benchmarks. For conducting the experiments, we employ SQL Server 2017 and Julia programming Lagrange 0.6.4. Also, to solve optimization models, we use CPLEX 12.9 and we employ PsiDB for processing each batch of queries in database. All the computational experiments are conducted on a workstation with 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 Input/Output (I/O) rate across different experimental runs. The workstation runs on Windows 10 Enterprise Version 1709.

6.2.1 Database Benchmark TPC-H

In this section, we employ the well-known TPC-H database benchmark for showing the effectiveness of the QBP and our proposed heuristics [55, 56]. TPC-H database represents a customer-supplier system. It contains information of the provided parts by the suppliers and the orders by the customers. This database benchmark takes a parameter, the so-called Scale Factor (SF), as an input. This parameter indicates the approximate size (in gigabytes) of the random TPC-H database that will be generated. In this section, we consider three different values for the SF, i.e., \{1, 5, 10\}. This implies that we randomly generate three TPC-H databases with the size of approximately 1, 5, and 10 gigabytes, respectively. The random databases have 5 tables and there are relationships between some of them as shown Figure 6.9. The relationships are defined by the so-called foreign keys and shown by arrows in Figure 6.9. These relationships are important because PsiDB uses them when processing a batch of queries. For example, if there is a batch of queries on only tables PART and SUPPLIER, PsiDB has to also use table PARTSUPP (when joining tables) as it contains both foreign keys of tables PART and SUPPLIER. As an aside, we note that since there are only 5 tables, in theory, there should be \(2^5 - 1 = 31\) scenarios for a query to involve the tables. However, because of the relationships between tables, only 17 scenarios are feasible. The number of records in each table for different values of SF are reported in Table 6.2.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>SF=1</th>
<th>SF=5</th>
<th>SF=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEITEM</td>
<td>6,001,215</td>
<td>30,006,075</td>
<td>60,012,150</td>
</tr>
<tr>
<td>ORDERS</td>
<td>1,500,000</td>
<td>7,500,000</td>
<td>15,000,000</td>
</tr>
<tr>
<td>PARTSUPP</td>
<td>800,000</td>
<td>4,000,000</td>
<td>8,000,000</td>
</tr>
<tr>
<td>PART</td>
<td>200,000</td>
<td>1,000,000</td>
<td>2,000,000</td>
</tr>
<tr>
<td>SUPPLIER</td>
<td>10,000</td>
<td>50,000</td>
<td>100,000</td>
</tr>
</tbody>
</table>

Table 6.2: The number of records in each table of TPC-H for three values of SF
We first attempt to estimate the coefficients of the batch processing time prediction function, i.e., $\beta = (\beta_0, \beta_1, \ldots, \beta_{m+1})$. Since each SF basically defines a new database, the coefficients should be estimated for each one independently. Hence, for each SF, we randomly generate 2754 data points, i.e., batches of queries. These data points are divided over 81 classes, each containing 34 batches, based on the number of queries that they involve. Specifically, the number of queries in each class of batches is taken from the set \{50, 75, 100, 125, \ldots, 2050\}. To create the batches of each class, we first generate two random scenarios for showing the percentage of the data that should be retrieved as the result of processing a batch. Each scenario can take a value from the interval \([0\%, 5\%]\). For each scenario, we created 17 batches. Specifically, as mentioned earlier, there are 17 possible ways of including tables in (a batch) queries for TPC-H databases. So, for each one, we randomly generate one batch of queries such that the amount of data that should be retrieved for it matches its associated scenario.

After creating random batches for each SF, we execute them individually and record their associated values for the dependent and independent variables of linear regression (see Section 5.3.1). We use 70% of data as the training set and the remaining 30% as the testing set. Overall, the results show that

$$\beta = (1.6032, 0.0023, 0.1228, 3.3580, 2.0077, 4.6639, 1.8580),$$

$$\beta = (10.8716, 0.0028, 0.2083, 5.5188, 3.3459, 7.8262, 3.1003),$$

and

$$\beta = (17.1205, 0.0028, 0.2613, 6.8871, 4.2800, 9.8678, 3.8612)$$

for SF=1, SF=5, and SF=10, respectively. The R-squared is around 0.95 for all the cases.

In order to show the effectiveness of solving the query batching problem, a total of 120 instances are generated for each SF. An instance includes a specific number of queries on SF 1, 5, or 10 of the TPC-H database. With SF taking three values 1, 5, and 10, a total of 360 instances
are generated in this experiment. The instances are divided over three classes (for each SF) of small, medium, and large instances. For the small-sized instances (denoted by S), the number of queries is taken form the set \{32, 64\}. For the medium-sized instances (denoted by M), the number of queries is taken form the set \{256, 512\}. Finally, for the large-sized instances (denoted by L), the number of queries is taken form the set \{2048, 4096\}. Therefore, each class of instances contains two sub-classes. For example, for Class S, the first subclass has instances with 32 queries each and the second subclass has instances with 64 queries each. Within each instance subclass, 20 random instances are generated. We note that since TPC-H simulates a customer-supplier system database in the real world, it comes with some suggested scenarios for generating a set of queries to make sense in practice (see the details in [55, 56]). Hence, when creating an instance, we follow the suggested scenarios. Specifically, to generate each instance, we first randomly select a subset of all 17 valid combinations of employing tables and then generate all randomly queries based on the selected combinations. Table 6.3 summarizes the performance of RCSA-I and RCSA-II on all 360 instances of this experiment. Numbers reported in this table for each instance subclass are averages over 20 instances.

In Table 6.3, the columns labeled ‘# of batches’ reports the number of batches obtained by using a heuristic; the columns labeled ‘Opt Time (s)’ show the solution time of a heuristic in seconds; the columns labeled ‘Total Time (s)’ show the total processing time of an instance (in seconds) which include the solution time of a heuristic (if used any) and the processing time of the prescribed batches by the model; the column labeled ”Default” contains the results for the default setting of PsiDB, which is a single batch containing all queries; Finally, the columns labeled ‘Imp (%)’ reports the percentage improvement in the total time obtained by using a heuristic to solve the QBP. As an aside, we note that PsiDB is shown to be faster than processing queries one by one by a factor of around 30 [32]. Consequently, we do not report any result for processing queries (of each batch) individually.
Table 6.3: Performance of RCSA-I and RCSA-II on TPC-H databases

<table>
<thead>
<tr>
<th>SF</th>
<th>n</th>
<th>Default</th>
<th>RCSA-I</th>
<th>RCSA-II</th>
<th>Imp (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total Time (s)</td>
<td># of batches</td>
<td>Opt Time (s)</td>
<td>Total Time (s)</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>75.0</td>
<td>2.4</td>
<td>0.6</td>
<td>41.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>130.9</td>
<td>2.3</td>
<td>0.2</td>
<td>83.1</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>306.5</td>
<td>2.6</td>
<td>0.2</td>
<td>182.6</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>414.6</td>
<td>1.9</td>
<td>0.2</td>
<td>319.9</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>756.7</td>
<td>2.2</td>
<td>4.3</td>
<td>550.2</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>964.4</td>
<td>2.1</td>
<td>13.1</td>
<td>745.1</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>453.5</td>
<td>2.1</td>
<td>0.2</td>
<td>268.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>710.2</td>
<td>2.1</td>
<td>0.1</td>
<td>453.7</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1548.1</td>
<td>2.4</td>
<td>0.4</td>
<td>964.3</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2335.7</td>
<td>2.0</td>
<td>0.4</td>
<td>1822.2</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>3772.3</td>
<td>2.1</td>
<td>1.9</td>
<td>2835.8</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>5176.5</td>
<td>2.3</td>
<td>9.4</td>
<td>3751.3</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>1120.8</td>
<td>2.1</td>
<td>0.1</td>
<td>702.6</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1638.4</td>
<td>2.3</td>
<td>0.1</td>
<td>969.5</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>3259.4</td>
<td>2.4</td>
<td>0.2</td>
<td>2057.7</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>4479.9</td>
<td>2.2</td>
<td>0.4</td>
<td>3192.3</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>8165.0</td>
<td>2.3</td>
<td>2.6</td>
<td>5484.4</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>10502.2</td>
<td>2.3</td>
<td>10.1</td>
<td>7505.9</td>
</tr>
</tbody>
</table>

The last two columns of the table reveal an improvement ranging from 21.0 to 45.0 % in the query processing time based on the proposed heuristic methods when compared to the default setting. Another interesting outcome from comparing the results of the two heuristics for large instances (in the table) is that their total times are close. This implies that the time that RCSA-II’s solution saves by a better batching is compromised by the additional time it spends in the optimization phase. For example, the last row of Table 6.3 shows that better batching of RCSA-II can barely compensate 176.3 seconds of extra time that it needs for optimization comparing to RCSA-I.

Figure 6.10 shows the total time of all instances of the subclasses \( n = 31 \) and \( n = 4096 \) for different SF values. In the Figure, white bars, gray bars, black bars represent the total time (in seconds) of the default setting, RCSA-I, and RCSA-II, respectively. Observe that batching the queries using the proposed heuristics reduces the total time except for the cases in which final recommendation of the optimization model is as same as the default setting (assigning all queries to one single batch).
Table 6.4 shows the (relative) optimality gap between the objective values reported by the heuristic methods and the global dual bounds reported by CPLEX when solving the exact MBQP formulation, on instances with \( n \leq 512 \) over a two-hour time limit. In this table, the column labelled 'Exact Opt Time (s)' shows the solution time of CPLEX for solving the exact MBQP. Again, numbers reported in this table for each instance subclass are averages over 20 instances. CPLEX was able to solve all instances with \( n \leq 256 \) and most instances with \( n = 512 \) to optimality within the imposed time limit. Among instances with \( n = 512 \), only two, three, and two of them were not solved to optimality for SF= 1, SF = 5, and SF = 10, respectively. From the table, we observe that RCSA-II was able to generate an optimal solution for all instances that their optimal solutions were known, i.e., solved within 2 hours time limit by CPLEX. Even for those few large instances that CPLEX was not able to prove the optimality within 2 hours, the RCSA-II relative optimality gap from the global dual bound is below 5%. This provide evidences that RCSA-II can at least provide near optimal solutions for such instances. Note that the results of RCSA-I are similar to RCSA-II but slightly worse for instances with \( n = 512 \). This explain the observation that we made earlier about the similarity of the total time of RCSA-I and RCSA-II for large instances in Table 6.3.

Table 6.4: Comparing the quality of the solutions obtained by the proposed heuristics with optimal solutions

<table>
<thead>
<tr>
<th>SF</th>
<th>n</th>
<th>Exact Opt Time (s)</th>
<th>RCSA-I gap (%)</th>
<th>RCSA-II gap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>0.4</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.2</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>324.1</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2463.0</td>
<td>2.54</td>
<td>2.45</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>0.6</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.9</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>309.3</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2887.8</td>
<td>3.37</td>
<td>3.37</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>0.5</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>392.9</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2143.2</td>
<td>2.14</td>
<td>2.14</td>
</tr>
</tbody>
</table>
6.2.2 Database Benchmark JOB

The other benchmark we use to test the effectiveness of the QBP and our proposed heuristics is the Join-Order Benchmark (JOB) [57]. JOB is a fixed-sized database with almost 4GB of data which is created based on the well-known Internet Movie Database (IMDB). It contains detailed information about movies and related facts about actors, directors, production companies, etc. This database contains 23 tables, and relations among them. All the tables in this database and their relationships are shown in Figure 6.11. Note that there were only 17 valid combinations for involving tables in the TPC-H benchmark, but for the JOB benchmark, the number of valid combinations are significantly larger. The total number of combinations is $2^{23} - 1$ but because of the relationships between the tables not all combinations are valid. So, in this section, we simply select 165 valid combinations and employ them for generating instances. These 165 valid combinations are obtained based on the suggested scenarios that come with the JOB benchmark for ensuring that queries make sense in practice (see the details in [57]).

Table 6.5: Performance of RCSA-I and RCSA-II on the JOB database

<table>
<thead>
<tr>
<th>n</th>
<th>Default</th>
<th>RCSA-I</th>
<th>RCSA-II</th>
<th>Imp (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Time (s)</td>
<td># of batches</td>
<td>Opt Time (s)</td>
<td>Total Time (s)</td>
</tr>
<tr>
<td>32</td>
<td>145.1</td>
<td>2.4</td>
<td>0.2</td>
<td>115.3</td>
</tr>
<tr>
<td>64</td>
<td>325.4</td>
<td>2.2</td>
<td>0.1</td>
<td>263.9</td>
</tr>
<tr>
<td>256</td>
<td>1031.0</td>
<td>2.4</td>
<td>0.3</td>
<td>836.5</td>
</tr>
<tr>
<td>512</td>
<td>1757.6</td>
<td>1.9</td>
<td>0.4</td>
<td>1408.3</td>
</tr>
<tr>
<td>2048</td>
<td>3694.3</td>
<td>1.8</td>
<td>1.6</td>
<td>3094.9</td>
</tr>
<tr>
<td>4096</td>
<td>4502.8</td>
<td>2.3</td>
<td>14.6</td>
<td>3394.0</td>
</tr>
</tbody>
</table>

Again we first start by estimating the coefficients of the batch processing time prediction function, i.e., $\beta$. We randomly generate 13,365 data points, i.e., batches of queries. We generate the data points for the JOB benchmark following a similar procedure that we used for the TPC-H benchmark. Specifically, the data points are divided over 81 classes, each containing 165 batches, based on the number of queries that they involve. The number of queries in each class of batches
is again taken from the set \( \{50, 75, 100, 125, \ldots, 2050\} \). For each class, we generate a single scenario for showing the percentage of the data that should be retrieved as the result of processing a batch. Each scenario can take a value from the interval \((0\%, 5\%)\). For each scenario, we created 165 batches. Specifically, there are 165 selected valid ways of including tables in (a batch) queries for the JOB databases. So, for each one, we randomly generate one batch of queries such that the amount of data that should be retrieved for it matches its associated scenario.

After creating random batches, we execute them individually and record their associated values for the dependent and independent variables of linear regression (see Section 5.3.1). We use 70\% of data as the training set and the remaining 30\% as the testing set. Overall, the results show an R-squared of around 0.98 can be obtained by setting,

\[
\beta = (2.3028, 0.0004, 0.0289, 0.5822, 0.4903, 0.5086, 0.5125, 0.5537, 0.5201, 0.4874, 0.4790, 0.5086, \\
0.5125, 0.5537, 0.5201, 0.4874, 0.4790, 0.4994, 0.4671, 0.4651, 0.4915, 0.4920, 0.4641, 0.4645, 0.4880, 0.4956, 0.4877, 0.4635, 0.4485, 0.4603).
\]

In order to show the effectiveness of solving the query batching problem on the JOB database, a total of 120 instances are generated. Similar to what we did for TPC-H database, the instances are divided over three classes of S, M, and L (based on their number of queries). Each class contains two subclasses, each with 20 instances. Table 6.5 shows the result of the proposed heuristics on the JOB database. Numbers reported in this table for each instance subclass are averages over 20 instances. The last two columns demonstrate that the proposed heuristics reduce the total time up to 24.6\%. Observe that since the JOB database contains more tables compared to the TPC-H databases, the performance of RCSA-I and RCSA-II are different. Although, RCSA-II performs better in small instances (in terms of the total time), RCSA-I outperforms RCSA-II by around 8.2\% for the largest subclass of instances. This can be mainly because of the significant increase in the Opt Time of RCSA-II. Figure 6.12 highlights this observation further because (similar to Figure 6.10) it shows the total time of all instances of all subclasses under default setting, RCSA-I,
and RCSA-II. Another interesting point from Table 6.5 is that, unlike the TPC-H databases, the number of batches for RCSA-II is smaller than the number of batches for RCSA-I. This outcome means that RCSA-II generates fewer batches by putting one table in several batches that in turn gains better objective values.

Table 5 shows the (relative) optimality gap between the objective values reported by the heuristic methods and the global dual bounds reported by CPLEX when solving the exact MBQP formulation on instances with \( n \leq 256 \) over a two-hour time limit. CPLEX was able to solve all instances with \( n = 32 \) and most instances with \( n = 64 \) and \( n = 256 \) to optimality within the imposed time limit. Among the instances with \( n = 64 \), only 2 instances and among the instances with \( n = 256 \), 8 instances were not solved to optimality. From the table, we observe that both RCSA-I and RCSA-II performed the same. They were able to generate an optimal solution for all instances that their optimal solutions were known, i.e., solved within 2 hours time limit by CPLEX. Even for those few large instances that CPLEX was not able to prove the optimality within 2 hours, both heuristics relative optimality gap from the global dual bound is below 11%.

Table 6.6: Comparing the quality of the solutions obtained by the proposed heuristics with optimal solutions

<table>
<thead>
<tr>
<th>( n )</th>
<th>Exact Opt Time (s)</th>
<th>RCSA-I gap (%)</th>
<th>RCSA-II gap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5.9</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>64</td>
<td>726.9</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>256</td>
<td>3938.2</td>
<td>10.10</td>
<td>10.10</td>
</tr>
</tbody>
</table>
Figure 6.7: Time for processing $Q$ and distribution under (a) SF=1, (b) SF=10, and (c) SF=100
Figure 6.8: Execution time of PsiDB and SQL Server under different database sizes and query numbers

Figure 6.9: The structure of a TPC-H database
Figure 6.10: Total time (in seconds) of all 20 instances of the smallest and largest subclasses (white bars: default setting, gray bars: RCSA-I, black bars: RCSA-II)
Figure 6.11: The structure of the JOB database
Figure 6.12: Total time (in seconds) of all 20 instances of different classes (white bars: default setting, gray bars: RCSA-I, black bars: RCSA-II)
Chapter 7: Conclusions

7.1 Final Remarks

In this paper, for the first time (to the best of our knowledge), we studied the query batching problem. This problem seeks to partition a given set of queries into some batches before retrieving them from a database system in order to minimize the total retrieving/processing time. This optimization problem is challenging because computing the time required for processing a given batch of queries is not a trivial task. We developed a generic three-phase approach for solving the problem for any given database system. In this first phase, our approach attempts to generate a quadratic function for predicting the processing time of any batch of queries for the given database. In the second phase, our approach uses the obtained quadratic function and develops a mixed binary quadratic programming formulation for the query batching problem accordingly. Finally, in the last phase, our proposed approach uses two custom-built heuristic approaches, i.e., RCSA-I and RCSA-II, for quickly solving the obtained formulation in practice. We tested our proposed approach on two well-known database benchmarks by conducting a comprehensive computational study. The results showed that a reduction of up to 45% is achievable for the total processing times in the database benchmarks when employing our proposed approach.

We hope that the simplicity, versatility, and performance of our proposed approach encourage practitioners/researchers to consider employing/developing effective query batching optimizers. There are several future research directions that can be considered for this study. One direction can be developing better exact or heuristic solution approaches for solving instances with larger number of queries. Alternatively, developing effective machine learning methods for predicting
the outcome, i.e., an optimal solution, of the proposed mixed binary quadratic program can be an interesting research direction too. Another research direction can be developing some theories and/or methodologies for identifying an optimal amount of time that one should wait to receive queries before starting to solve the query batching problem.

7.2 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


Appendix A: Proof of Query Batching Problem is an NP-Hard Problem

The query batching problem when employing the proposed processing time prediction function (A.1) is NP-hard.

\[ P(S') := \left[ \beta_0 + \beta_1 n' + \sum_{t \in T} \beta_{t+1} \log_2(s_t)y'_t \right]^2, \quad (A.1) \]

**Proof.** In order to prove that the QBP is NP-hard, it is sufficient to show that its corresponding decision problem, QBP, is NP-complete. Note that QBP is obviously in NP and an instance of QBP can be shown by

\[ [m, n, U, \beta_0, \beta_1, \beta_2 \log_2(s_1), \ldots, \beta_{m+1} \log_2(s_m), Q_1, \ldots, Q_m]. \]

In the remaining, we show that Partition-Problem is polynomially reducible to QBP, where Partition-Problem is defined as follows: given a set \( \{c_1, \ldots, c_m\} \) of positive integers, does there exist a subset \( A \subseteq \{1, \ldots, m\} \) such that

\[ \sum_{i \in A} c_i = \sum_{i \in A \setminus \{1, \ldots, m'\}} c_i. \]

Observe that an integer programming formulation for Partition-Problem can be stated as follows (the second constraint is redundant but we keep it for the sake of the proof),

96
Now, given an arbitrary instance of Partition-Problem, denoted by \([m, c_1, \ldots, c_m']\), we construct an instance of QBP by setting,

- \(m = n = m'\)
- \(U = 0\)
- \(\beta_0 = -\frac{\sum_{i=1}^{m'} c_i}{2}\)
- \(\beta_1 = 0\)
- \(\beta_{t+1} \log(s_t) = c_t\) for all \(t \in \{1, \ldots, m'\}\)
- \(Q_t = \{t\}\) for all \(t \in \{1, \ldots, m'\}\)

Observe that the created instance of QBP is valid in a sense that it can be constructed in polynomial time and each of its queries involves at least one table. Now, we show that an instance of Partition-Problem, \([m, c_1, \ldots, c_m']\), is a Yes-instance if and only if its corresponding QBP instance is a Yes-instance. For doing so, we use the formulation proposed for the QBP and we show that when we plug the constructed instance into it, the formulation will be simplified/equivalent to the formulation of Partition-Problem.
In light of the above, observe that because \( \beta_1 = 0 \) and \( n_j = \sum_{i=1}^{n} x_{ij} \), the formulation of QBP can be simplified to,

\[
\sum_{j=1}^{n} \left[ \beta_0 z_j + \sum_{t \in T} \beta_{t+1} \log_2(s_i) y_{jt} \right]^2 \leq U
\]

\[
y_{jt} \leq \sum_{i \in Q_t} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
z_j \leq \sum_{i=1}^{n} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\}
\]

\[
\sum_{i \in Q_t} x_{ij} \leq n y_{jt} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
\sum_{t \in T} y_{jt} \leq m z_j \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
x_{ij}, y_{jt}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, n\}, \forall t \in T.
\]

Moreover since \( Q_t = \{t\} \) for all \( t \in \{1, \ldots, m'\} \) the term \( \sum_{i \in Q_t} x_{ij} \) is equivalent to \( x_{ij} \). So, the formulation can be simplified to,

\[
\sum_{j=1}^{n} \left[ \beta_0 z_j + \sum_{t \in T} \beta_{t+1} \log_2(s_i) y_{jt} \right]^2 \leq U
\]

\[
y_{jt} \leq x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
z_j \leq \sum_{i=1}^{n} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}
\]

\[
\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\}
\]

\[
x_{ij} \leq n y_{jt} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T
\]

\[
\sum_{t \in T} y_{jt} \leq m z_j \quad \forall j \in \{1, 2, \ldots, n\}
\]
\[ x_{ij}, y_{jt}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, n\}, \forall t \in T. \]

Note that a constraint of the form \( x_{ij} \leq n y_{jt} \) is equivalent to \( x_{ij} \leq y_{jt} \) since both \( x_{ij} \) and \( y_{ij} \) are binary variables. Therefore the formulation simplifies to,

\[
\sum_{j=1}^{n} \left[ \beta_0 z_j + \sum_{t \in T} \beta_{t+1} \log_2(s_i) y_{jt} \right]^2 \leq U
\]

\[ y_{jt} = x_{ij} \quad \forall j \in \{1, 2, \ldots, n\}, \forall t \in T \]

\[ z_j \leq \sum_{i=1}^{n} x_{ij} \quad \forall j \in \{1, 2, \ldots, n\} \]

\[ \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, n\} \]

\[ \sum_{t \in T} y_{jt} \leq m z_j \quad \forall j \in \{1, 2, \ldots, n\} \]

Consequently, the decision variable \( y_{jt} \) can be replaced by \( x_{ij} \) since \( m = n = m' \). So, the formulation can be simplified further to,

\[
\sum_{j=1}^{m'} \left[ \beta_0 z_j + \sum_{i=1}^{m'} \beta_{i+1} \log_2(s_i) x_{ij} \right]^2 \leq U
\]

\[ z_j \leq \sum_{i=1}^{m'} x_{ij} \quad \forall j \in \{1, 2, \ldots, m'\} \]

\[ \sum_{j=1}^{m'} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, m'\} \]

\[ \sum_{i=1}^{m'} x_{ij} \leq m' z_j \quad \forall j \in \{1, 2, \ldots, m'\} \]

\[ x_{ij}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, m'\}, \forall j \in \{1, 2, \ldots, m'\}. \]
Now, since $U = 0$, $\beta_0 = -\frac{\sum_{i=1}^{m'} c_i}{2}$, and $\beta_{t+1} \log(s_t) = c_t$ for all $t \in \{1, \ldots, m'\}$, the formulation can be rewritten as,

\[
\sum_{j=1}^{m'} \left[ -\frac{\sum_{i=1}^{m'} c_i}{2} z_j + \sum_{i=1}^{m'} c_i x_{ij} \right]^2 \leq 0
\]

\[
z_j \leq \sum_{i=1}^{m'} x_{ij} \quad \forall j \in \{1, 2, \ldots, m'\}
\]

\[
\sum_{j=1}^{m'} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, m'\}
\]

\[
\sum_{i=1}^{m'} x_{ij} \leq m' z_j \quad \forall j \in \{1, 2, \ldots, m'\}
\]

\[
x_{ij}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, m'\}, \forall j \in \{1, 2, \ldots, m'\}.
\]

Since the sum of squares is not positive, each square must be zero. So, the formulation can be simplified to,

\[
-\frac{\sum_{i=1}^{m'} c_i}{2} z_j + \sum_{i=1}^{m'} c_i x_{ij} = 0 \quad \forall j \in \{1, 2, \ldots, m'\}
\]  
(A.2)

\[
z_j \leq \sum_{i=1}^{m'} x_{ij} \quad \forall j \in \{1, 2, \ldots, m'\}
\]  
(A.3)

\[
\sum_{j=1}^{m'} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, m'\}
\]  
(A.4)

\[
\sum_{i=1}^{m'} x_{ij} \leq m' z_j \quad \forall j \in \{1, 2, \ldots, m'\}
\]  
(A.5)

\[
x_{ij}, z_j \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, m'\}, \forall j \in \{1, 2, \ldots, m'\}.
\]  
(A.6)

Observe that Constraints (A.3) and (A.5) guarantee that $z_j = 1$ if and only if $\sum_{i=1}^{m'} x_{ij} > 0$. This combined with Constraint (A.2) ensure that for any $j \in \{1, \ldots, m'\}$, $\sum_{i=1}^{m'} c_i x_{ij} = \frac{\sum_{i=1}^{m'} c_i}{2}$ if and only if $z_j = 1$ (or $\sum_{i=1}^{m'} x_{ij} > 0$). This itself combined with Constraint (A.4), which is defined for
partitioning the set of queries, guarantee that (if the instance is feasible then) there will be exactly two batches $j, j' \in \{1, \ldots, m'\}$ with $j \neq j'$ such that $\sum_{i=1}^{m'} c_i x_{ij} = \frac{\sum_{i=1}^{m'} c_i}{2}$ and $\sum_{i=1}^{m'} c_i x_{ij'} = \frac{\sum_{i=1}^{m'} c_i}{2}$. Note that the remaining batches must be empty since by assumptions $c_1, \ldots, c_{m'} > 0$. So, the above formulation is equivalent to,

$$-\frac{\sum_{i=1}^{m'} c_i}{2} + \sum_{i=1}^{m'} c_i x_{ij} = 0 \quad \forall j \in \{1, 2\}$$

$$\sum_{j=1}^{2} x_{ij} = 1 \quad \forall i \in \{1, 2, \ldots, m'\}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, m'\}, \forall j \in \{1, 2\}$$

This itself is equivalent to,

$$-\frac{\sum_{i=1}^{m'} c_i}{2} + \sum_{i=1}^{m'} c_i x_i = 0$$

$$-\frac{\sum_{i=1}^{m'} c_i}{2} + \sum_{i=1}^{m'} c_i (1 - x_i) = 0$$

$$x_i \in \{0, 1\} \quad \forall i \in \{1, 2, \ldots, m'\}$$

This is precisely an integer programming formulation of Partition-Problem. Therefore, the result follows.