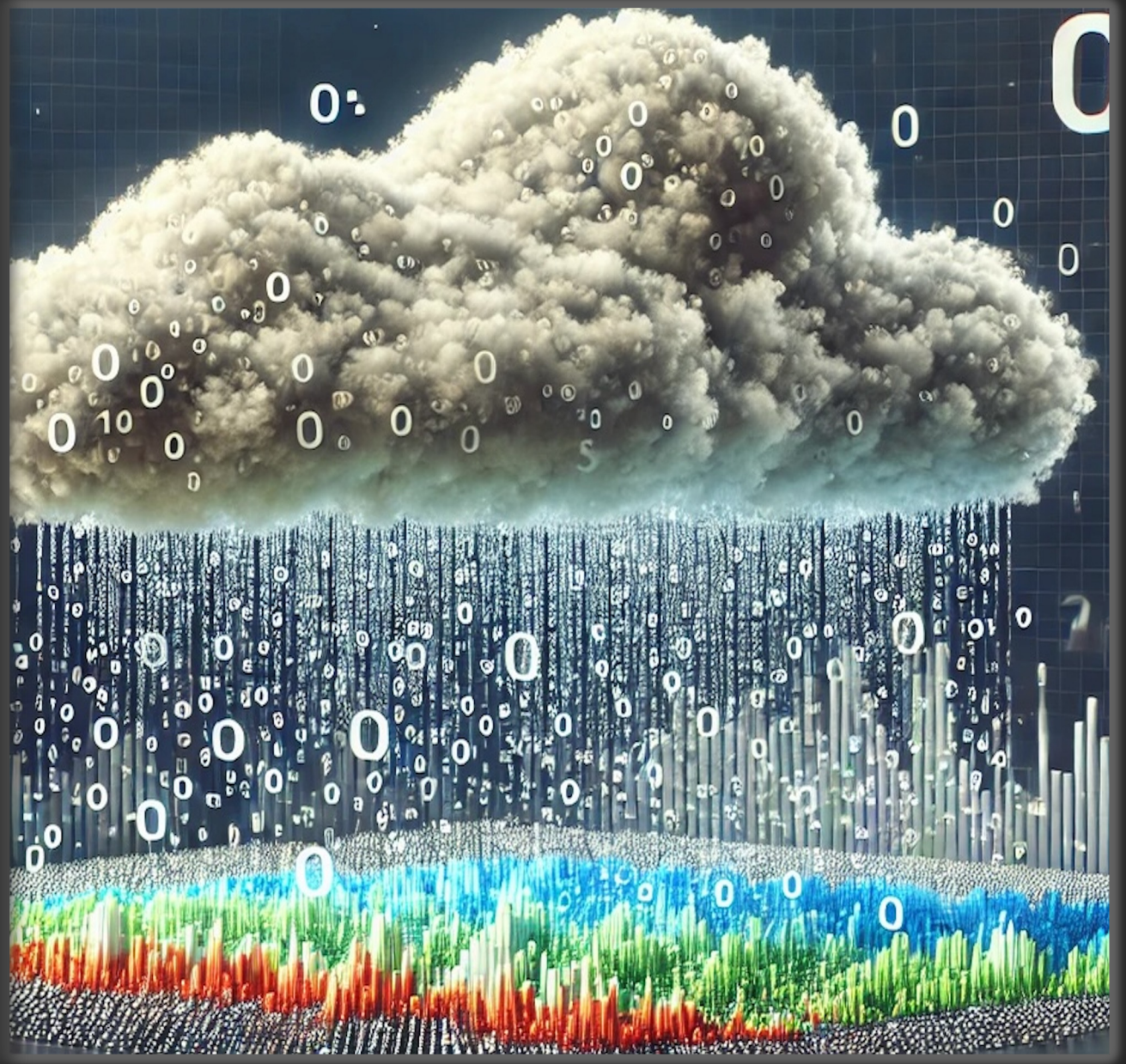
Carnegie Mellon University

Advanced Database Systems (15-721)

Lecture #03

The Extensible Cascades/ Volcano Query Optimizer



Fall 2024 Prof. Jignesh Patel

Announcement

- Join Piazza.
- Class notes schedule has been updated. Check here.
- Snowflake talk on Tue, Sept 10, noon-1 pm in GHC 6501. Also, free lunch!
- DB IAP on Mon 9/16 in GHC 4405 from 9am–3pm, in GHC 6101 after 3pm. To meet the companies, fill this form.
- Add your resumes to this form (and the associated drive).

BACKDROP

- Selinger's QO was widely adopted.
- A new direction was extensibility.
  - Want to add new types/objects and manage them in the database engine.
  - With all the good things that database offers, including declarative query processing and transaction management.
- New QO "rules" were being discovered.
- It was cumbersome to add these new rules to the Selinger-style QO.

The EXODUS Extensible DBMS Project: An Overview

Michael J. Carey, David J. DeWitt,
Goetz Graefe, David M. Haight,
Joel E. Richardson, Daniel T. Schuh,
Eugene J. Shekita, and Scott L. Vandenberg

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper presents an overview of EXODUS, an extensible database system project that is addressing data management problems posed by a variety of challenging new applications. The goal of the project is to facilitate the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager. In addition, it provides an architectural framework for building application-specific database systems; powerful tools to help automate the generation of such systems, including a rule-based query optimizer generator and a persistent programming language; and libraries of generic software components (e.g., access methods) that are likely to be useful for many application domains. We briefly describe each of the components of EXODUS in this paper, and we also describe a next-generation DBMS that we are now building using the EXODUS tools.

1. INTRODUCTION

Until fairly recently, research and development efforts in the database systems area have focused primarily on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, such as computer-aided design and manufacturing, scientific and statistical applications, large-scale AI systems, and image/voice applications, has now emerged as an important research direction. Such new applications differ from conventional database applications and from each other in a number of important ways. First of all, their data modeling requirements vary widely. The kinds of entities and relationships relevant to a VLSI circuit design are quite different from those of a banking application. Second, each new application area has a different, specialized set of operations that must be efficiently supported by the database system. For example, it makes little sense to talk about doing joins between satellite images. Efficient support for such specialized operations also requires new types of storage structures and access methods. For applications like VLSI design, involving spatial objects, R-Trees [Gutt84] are a useful access method for data storage and manipulation; to manage image data efficiently, the database system needs to provide large arrays as a basic data type. Finally, a number of new application areas require support for multiple versions of their entities [Snod85, Daya86, Katz86].

A number of research projects are addressing the needs of new applications by developing approaches to making a database system *extensible* [DBE87]. These projects include EXODUS[1] at the University of Wisconsin [Care86a, Carey86c], PROBE at CCA [Daya86, Mano86], POSTGRES at UC Berkeley [Ston86b, Rowe87], STARBURST at IBM Almaden Research Center [Schw86, Lind87], and GENESIS at the University of Texas-Austin [Bato88a, Bato88b]. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, their overall approaches are quite different. For example, POSTGRES is a complete

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant IRI-8657323, by IBM through two Fellowships, by DEC through its Incentives for Excellence program, and by donations from Apple Corporation, GTE Laboratories, the Microelectronics and Computer Technology Corporation (MCC), and Texas Instruments.

[1] EXODUS: A departure; in this case, from traditional approaches to database management. Also an **EX**tensible **O**bject-oriented **D**atabase **S**ystem.

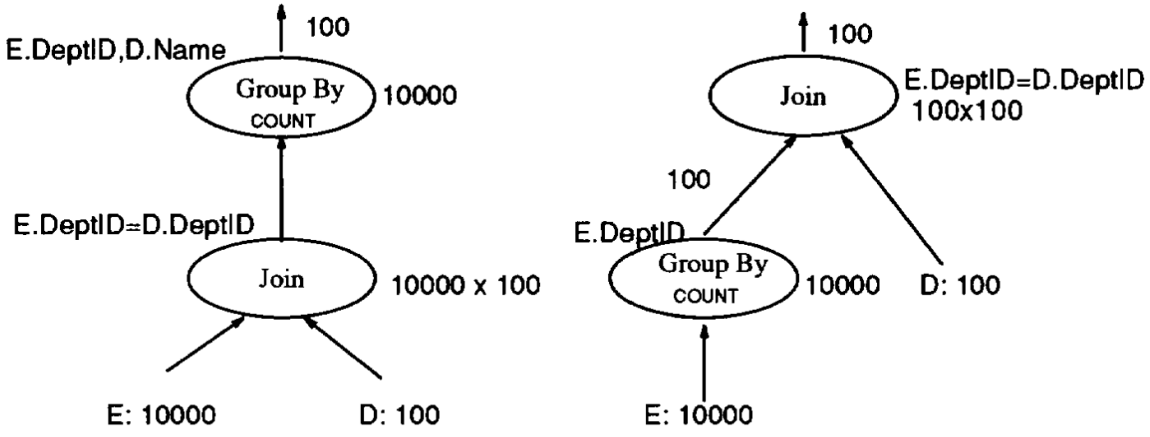
Rules

- $\sigma_{P1}(\sigma_{P2}(R)) \equiv \sigma_{P2}(\sigma_{P1}(R))$ (σ commutativity)
- $\sigma_{P1 \wedge P2 \ldots \wedge Pn}(R) \equiv \sigma_{P1}(\sigma_{P2}(\ldots \sigma_{Pn}(R)))$ (cascading σ)
- $\prod_{a1}(R) \equiv \prod_{a1}(\prod_{a2}(\ldots \prod_{ak}(R)\ldots))$, $a_i \subseteq a_{i+1}$ (cascading ∏)
- $R \bowtie S \equiv S \bowtie R$ (join commutativity)
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (join associativity)
- $\sigma_P(R \times S) \equiv (R \bowtie_P S)$, if P is a join predicate
- $\sigma_P(R \times S) \equiv \sigma_{P1}(\sigma_{P2}(R) \bowtie_{P4} \sigma_{P3}(S))$, where $P = p1 \wedge p2 \wedge p3 \wedge p4$
- $\prod_{A1,A2,\ldots An}(\sigma_P(R)) \equiv \prod_{A1,A2,\ldots An}(\sigma_P(\prod_{A1,\ldots An,\, B1,\ldots BM} R))$, where B1 … BM are columns in P

EXAMPLE OF A NEW RULE: GROUP BY BEFORE A JOIN

```
Emp(EmpID, LastName, FirstName, DeptID)
Dept (DeptID, Name)

SELECT D.DeptID, D.Name, COUNT(E.EmpID)
FROM Employee E, Department D
WHERE E.DeptID = D.DeptID
GROUP BY D.DeptID, D.Name
```



Performing Group-By before Join

Weipeng P. Yan Per-Åke Larson
Department of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
email: {pwyan,palarson}@bluebox.uwaterloo.ca

Abstract

Assume that we have an SQL query containing joins and a group-by. The standard way of evaluating this type of query is to first perform all the joins and then the group-by operation. However, it may be possible to perform the group-by early, that is, to push the group-by operation past one or more joins. Early grouping may reduce the query processing cost by reducing the amount of data participating in joins. We formally define the problem, adhering strictly to the semantics of NULL and duplicate elimination in SQL2, and prove necessary and sufficient conditions for deciding when this transformation is valid. In practice, it may be expensive or even impossible to test whether the conditions are satisfied. Therefore, we also present a more practical algorithm that tests a simpler, sufficient condition. This algorithm is fast and detects a large subclass of transformable queries.

1 Introduction

SQL queries containing joins and group-by are fairly common. The standard way of evaluating such a query is to perform all joins first and then the group-by operation. However, it may be possible to interchange the evaluation order, that is, to push the group-by operation past one or more joins.

Example 1 : Assume that we have the two tables:

```
Employee(EmpID, LastName, FirstName, DeptID)
Department(DeptID, Name)
```

EmpID is the primary key in the Employee table and DeptID is the primary key of Department. Each Employee row references the department (DeptID) to which the employee belongs. Consider the following query:

```
SELECT    D.DeptID, D.Name, COUNT(E.EmpID)
FROM      Employee E, Department D
WHERE     E.DeptID = D.DeptID
GROUP BY  D.DeptID, D.Name
```

Plan 1 in Figure 1 illustrates the standard way of evaluating the query: fetch the rows in tables E and D, perform the join, and group the result by D.DeptID and D.Name, while at the same time counting the number of rows in each group. Assuming that there are 10000 employees and 100 departments, the input to the join is 10000 Employee rows and 100 Department rows and the input to the group-by consists of 10000 rows. Now consider Plan 2 in Figure 1. We first group the Employee table DeptID and perform the COUNT, then join the resulting 100 rows to the 100 Department rows. This reduces the join from (10000×100) to (100×100). The input cardinality of the group-by remains the same, resulting in an overall reduction of query processing time. □

In the above example, it was both possible and advantageous to perform the group-by operation before the join. However, it is also easy to find examples where this is (a) *not* possible or (b) possible but *not* advantageous. This raises the following general questions:

1. Exactly under what conditions is it possible to perform a group-by operation before a join?
2. Under what conditions does this transformation reduce the query processing cost?

This paper concentrates on answering the first question. Our main theorem provides sufficient and necessary conditions for deciding when this transformation is valid. The conditions cannot always be tested efficiently so we also propose a more practical algorithm which tests a simpler, sufficient condition.

The rest of the paper is organized as follows. Section 2 summarizes related research work. Section 3 defines the class of queries that we consider. Section 4 presents the formalism that our results are based on. Section 4.1 presents an SQL2 algebra whose operations

ANOTHER EXAMPLE: OUTERJOINS

- Not associative or commutative like inner joins.
  - Nulls make it complicated.
- Similarly, issues with anti-joins
  - Only inner joins and full outer joins are commutative.

Also need robust ways to deal with materialized views and CTEs.

R

tid	a
r1	1
r2	3
r3	5

S

tid	a	b
s1	1	1
s2	1	2
s3	3	3
s4	3	4

T

tid	b
t1	1

Figure 1: Contents of table R, S and T

r1	s1	t1
r2	-	-
r3	-	-

(a) $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.b=T.b}{\bowtie} T)$

r1	s1	t1

(b) $(R \overset{R.a=S.a}{\longrightarrow} S) \overset{S.b=T.b}{\bowtie} T$

Figure 2: Query Results (- represents a null value)

Jun Rao, Bruce G. Lindsay, Guy M. Lohman, Hamid Pirahesh, David E. Simmen: Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering. ICDE 2001: 585-594

BACKDROP

- The "rule book" was growing.
  - One could enhance the System R optimizer (e.g., expand the search space to look for bushy plans), but the code change was cumbersome.
  - Can we get use abstractions in the query optimizer code?
- Queries were getting more complex (more joins).
- The traditional System R optimization method finds an optimal plan only at the "end."
  - Would like to find a "reasonable" plan earlier, and stop the QO when a "good enough" plan is found.

The EXODUS Extensible DBMS Project: An Overview

Michael J. Carey, David J. DeWitt,
Goetz Graefe, David M. Haight,
Joel E. Richardson, Daniel T. Schuh,
Eugene J. Shekita, and Scott L. Vandenberg

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper presents an overview of EXODUS, an extensible database system project that is addressing data management problems posed by a variety of challenging new applications. The goal of the project is to facilitate the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager. In addition, it provides an architectural framework for building application-specific database systems; powerful tools to help automate the generation of such systems, including a rule-based query optimizer generator and a persistent programming language; and libraries of generic software components (e.g., access methods) that are likely to be useful for many application domains. We briefly describe each of the components of EXODUS in this paper, and we also describe a next-generation DBMS that we are now building using the EXODUS tools.

1. INTRODUCTION

Until fairly recently, research and development efforts in the database systems area have focused primarily on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, such as computer-aided design and manufacturing, scientific and statistical applications, large-scale AI systems, and image/voice applications, has now emerged as an important research direction. Such new applications differ from conventional database applications and from each other in a number of important ways. First of all, their data modeling requirements vary widely. The kinds of entities and relationships relevant to a VLSI circuit design are quite different from those of a banking application. Second, each new application area has a different, specialized set of operations that must be efficiently supported by the database system. For example, it makes little sense to talk about doing joins between satellite images. Efficient support for such specialized operations also requires new types of storage structures and access methods. For applications like VLSI design, involving spatial objects, R-Trees [Gutt84] are a useful access method for data storage and manipulation; to manage image data efficiently, the database system needs to provide large arrays as a basic data type. Finally, a number of new application areas require support for multiple versions of their entities [Snod85, Daya86, Katz86].

A number of research projects are addressing the needs of new applications by developing approaches to making a database system *extensible* [DBE87]. These projects include EXODUS[1] at the University of Wisconsin [Care86a, Carey86c], PROBE at CCA [Daya86, Mano86], POSTGRES at UC Berkeley [Ston86b, Rowe87], STARBURST at IBM Almaden Research Center [Schw86, Lind87], and GENESIS at the University of Texas-Austin [Bato88a, Bato88b]. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, their overall approaches are quite different. For example, POSTGRES is a complete

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant IRI-8657323, by IBM through two Fellowships, by DEC through its Incentives for Excellence program, and by donations from Apple Corporation, GTE Laboratories, the Microelectronics and Computer Technology Corporation (MCC), and Texas Instruments.

[1] EXODUS: A departure; in this case, from traditional approaches to database management. Also an **EX**tensible **O**bject-oriented **D**atabase **S**ystem.

BACKDROP

The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe
Portland State University
graefe@cs.pdx.edu

William J. McKenna
University of Colorado at Boulder
bill@cs.colorado.edu

Abstract

Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EXODUS optimizer generator prototype, the search engine is more extensible and powerful; it provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational select-project-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimization systems, it provides complete data model independence and more natural extensibility.

1. Introduction

While extensibility is an important goal and requirement for many current database research projects and system prototypes, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application domains far beyond the capabilities of most existing database systems. Second, in order to overcome acceptance problems in emerging database application areas such as scientific computation, database systems must achieve at least the same performance as the file systems currently in use. Additional software layers for database management must be counterbalanced by database performance advantages normally not used in these application areas. Optimization and parallelization are prime candidates to provide these performance advantages, and tools and techniques for optimization and parallelization are crucial for the wider use of extensible database technology.

For a number of research projects, namely the Volcano extensible, parallel query processor [4], the REVELATION OODBMS project [11] and optimization and parallelization in scientific databases [20] as well as to assist research efforts by other researchers, we have built a new extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive; while it had proven the feasibility and validity of the optimizer generator paradigm, it was difficult to construct efficient, production-quality optimizers. Therefore, we designed a new optimizer generator, requiring several important improvements over the EXODUS prototype. First, this new optimizer generator had to be usable both in the Volcano project with the existing query execution software as well as in other projects as a stand-alone tool. Second, the new system had to be more efficient, both in optimization time and in memory consumption for the search. Third, it had to provide effective, efficient, and extensible support for physical properties such as sort order and compression status. Fourth, it had to permit use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Finally, it had to support flexible cost models that permit generating dynamic plans for incompletely specified queries.

In this paper, we describe the Volcano Optimizer Generator, which will soon fulfill all the requirements above. Section 2 introduces the main concepts of the Volcano optimizer generator and enumerates facilities for tailoring a new optimizer. Section 3 discusses the optimizer search strategy in detail. Functionality, extensibility, and search efficiency of the EXODUS and Volcano optimizer generators are compared in Section 4. In Section 5, we describe and compare other research into extensible query optimization. We offer our conclusions from this research in Section 6.

2. The Outside View of the Volcano Optimizer Generator

In this section, we describe the Volcano optimizer generator as seen by the person who is implementing a database system and its query optimizer. The focus is the wide array of facilities given to the optimizer implementor, i.e., modularity and extensibility of the Volcano optimizer generator design. After considering the design principles of the Volcano optimizer generator, we discuss generator input and operation. Section 3 discusses the search strategy used by optimizers generated with the Volcano optimizer generator.

Figure 1 shows the optimizer generator paradigm. When the DBMS software is being built, a model specification is translated into optimizer source code, which is then compiled and linked with the other DBMS

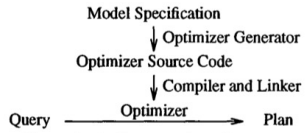


Figure 1. The Generator Paradigm.

The Cascades Framework for Query Optimization

Goetz Graefe

Abstract

This paper describes a new extensible query optimization framework that resolves many of the shortcomings of the EXODUS and Volcano optimizer generators. In addition to extensibility, dynamic programming, and memorization based on and extended from the EXODUS and Volcano prototypes, this new optimizer provides (i) manipulation of operator arguments using rules or functions, (ii) operators that are both logical and physical for predicates etc., (iii) schema-specific rules for materialized views, (iv) rules to insert "enforcers" or "glue operators," (v) rule-specific guidance, permitting grouping of rules, (vi) basic facilities that will later permit parallel search, partially ordered cost measures, and dynamic plans, (vii) extensive tracing support, and (viii) a clean interface and implementation making full use of the abstraction mechanisms of C++. We describe and justify our design choices for each of these issues. The optimizer system described here is operational and will serve as the foundation for new query optimizers in Tandem's NonStop SQL product and in Microsoft's SQL Server product.

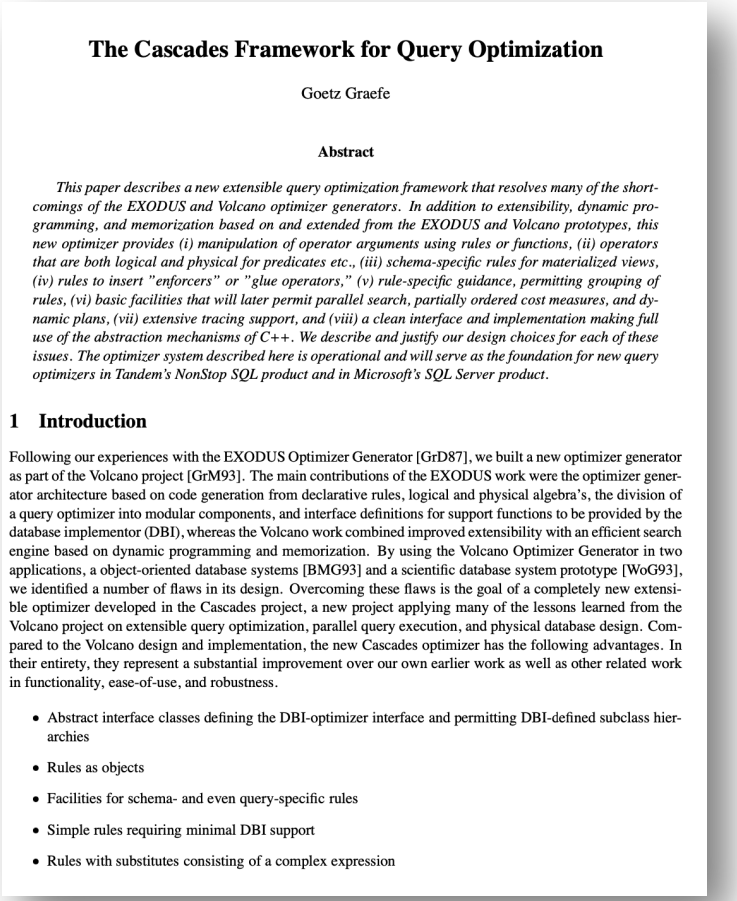
1 Introduction

Following our experiences with the EXODUS Optimizer Generator [GrD87], we built a new optimizer generator as part of the Volcano project [GrM93]. The main contributions of the EXODUS work were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebra's, the division of a query optimizer into modular components, and interface definitions for support functions to be provided by the database implementor (DBI), whereas the Volcano work combined improved extensibility with an efficient search engine based on dynamic programming and memorization. By using the Volcano Optimizer Generator in two applications, a object-oriented database systems [BMG93] and a scientific database system prototype [WoG93], we identified a number of flaws in its design. Overcoming these flaws is the goal of a completely new extensible optimizer developed in the Cascades project, a new project applying many of the lessons learned from the Volcano project on extensible query optimization, parallel query execution, and physical database design. Compared to the Volcano design and implementation, the new Cascades optimizer has the following advantages. In their entirety, they represent a substantial improvement over our own earlier work as well as other related work in functionality, ease-of-use, and robustness.

- Abstract interface classes defining the DBI-optimizer interface and permitting DBI-defined subclass hierarchies
- Rules as objects
- Facilities for schema- and even query-specific rules
- Simple rules requiring minimal DBI support
- Rules with substitutes consisting of a complex expression

STATE MANAGEMENT

- In bottom-up (also called construction-based) approach, one can “forget” subplans from the past
  - Only keep the cheapest or “interesting” plans.
  - So, for the nth join, we just need to know the best + interesting n-1 join subplans to proceed.
  - Can forget about the n-2, n-3, … join subplans that don’t contribute to the best + interesting n-1 plans.
- In top-down (also called transformation-based), we will walk all over the plan space.
  - Now, we have to remember all the subplans we have visited.
  - Need a compact data structure to represent sub-plans: Memoization!



The Cascades Framework for Query Optimization

Goetz Graefe

Abstract

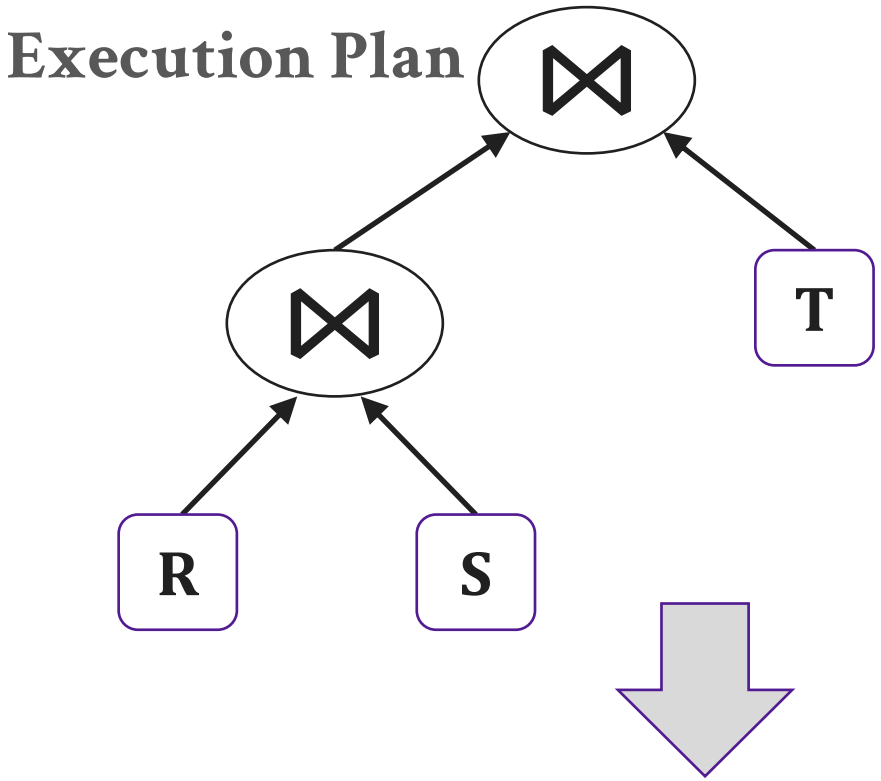
This paper describes a new extensible query optimization framework that resolves many of the shortcomings of the EXODUS and Volcano optimizer generators. In addition to extensibility, dynamic programming, and memorization based on and extended from the EXODUS and Volcano prototypes, this new optimizer provides (i) manipulation of operator arguments using rules or functions, (ii) operators that are both logical and physical for predicates etc., (iii) schema-specific rules for materialized views, (iv) rules to insert "enforcers" or "glue operators," (v) rule-specific guidance, permitting grouping of rules, (vi) basic facilities that will later permit parallel search, partially ordered cost measures, and dynamic plans, (vii) extensive tracing support, and (viii) a clean interface and implementation making full use of the abstraction mechanisms of C++. We describe and justify our design choices for each of these issues. The optimizer system described here is operational and will serve as the foundation for new query optimizers in Tandem's NonStop SQL product and in Microsoft's SQL Server product.

1 Introduction

Following our experiences with the EXODUS Optimizer Generator [GrD87], we built a new optimizer generator as part of the Volcano project [GrM93]. The main contributions of the EXODUS work were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebra's, the division of a query optimizer into modular components, and interface definitions for support functions to be provided by the database implementor (DBI), whereas the Volcano work combined improved extensibility with an efficient search engine based on dynamic programming and memorization. By using the Volcano Optimizer Generator in two applications, a object-oriented database systems [BMG93] and a scientific database system prototype [WoG93], we identified a number of flaws in its design. Overcoming these flaws is the goal of a completely new extensible optimizer developed in the Cascades project, a new project applying many of the lessons learned from the Volcano project on extensible query optimization, parallel query execution, and physical database design. Compared to the Volcano design and implementation, the new Cascades optimizer has the following advantages. In their entirety, they represent a substantial improvement over our own earlier work as well as other related work in functionality, ease-of-use, and robustness.

- Abstract interface classes defining the DBI-optimizer interface and permitting DBI-defined subclass hierarchies
- Rules as objects
- Facilities for schema- and even query-specific rules
- Simple rules requiring minimal DBI support
- Rules with substitutes consisting of a complex expression

THE MEMO STRUCTURE

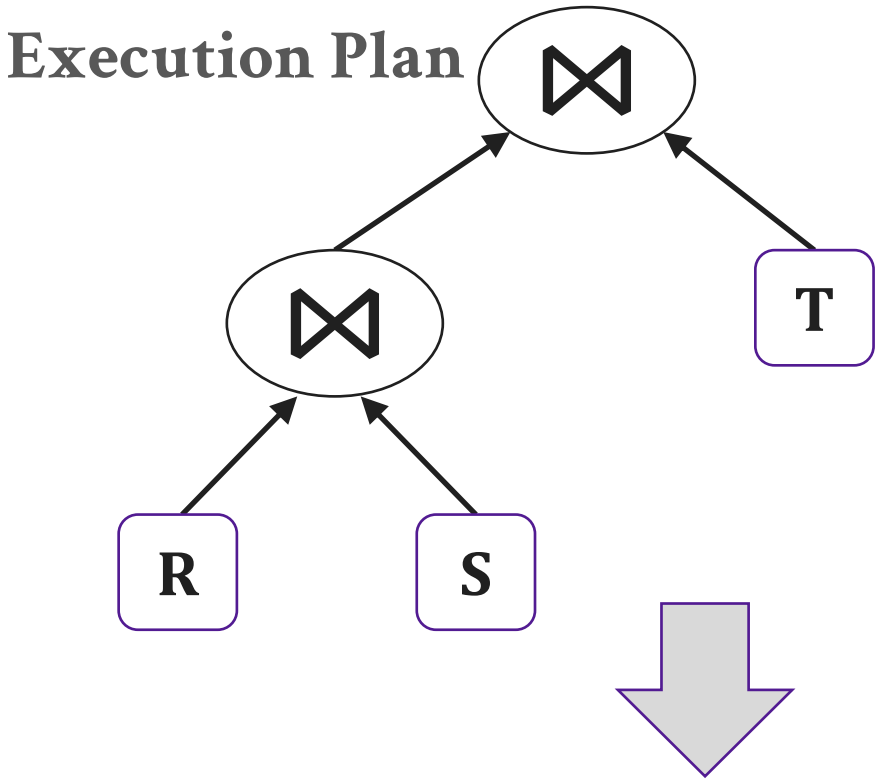


Join Graph (usually a hypergraph)

Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

THE MEMO STRUCTURE

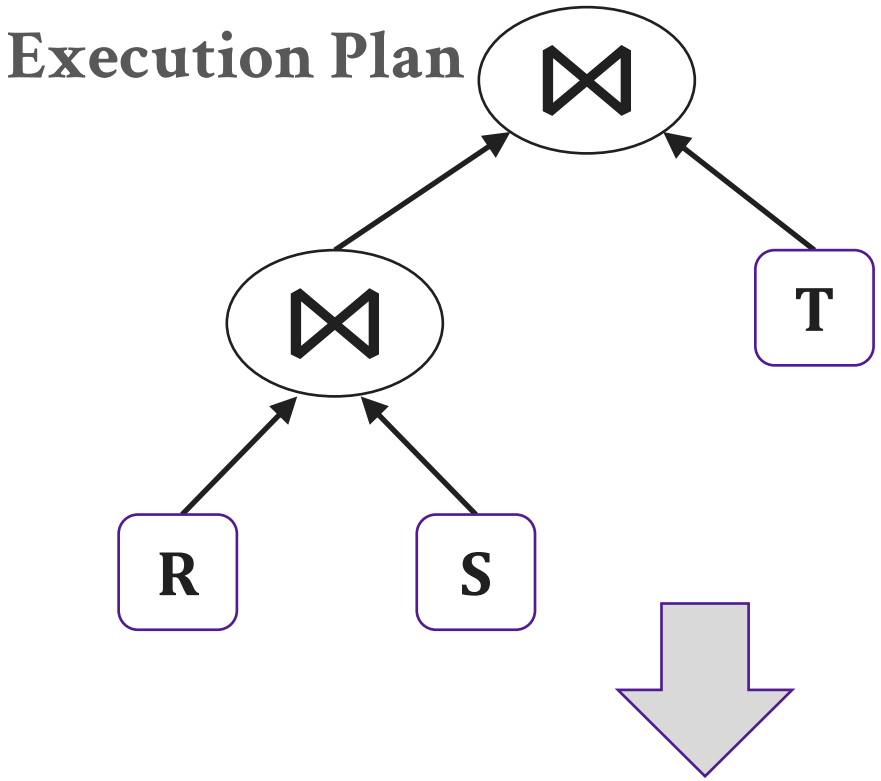


Join Graph (usually a hypergraph)

Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

THE MEMO STRUCTURE

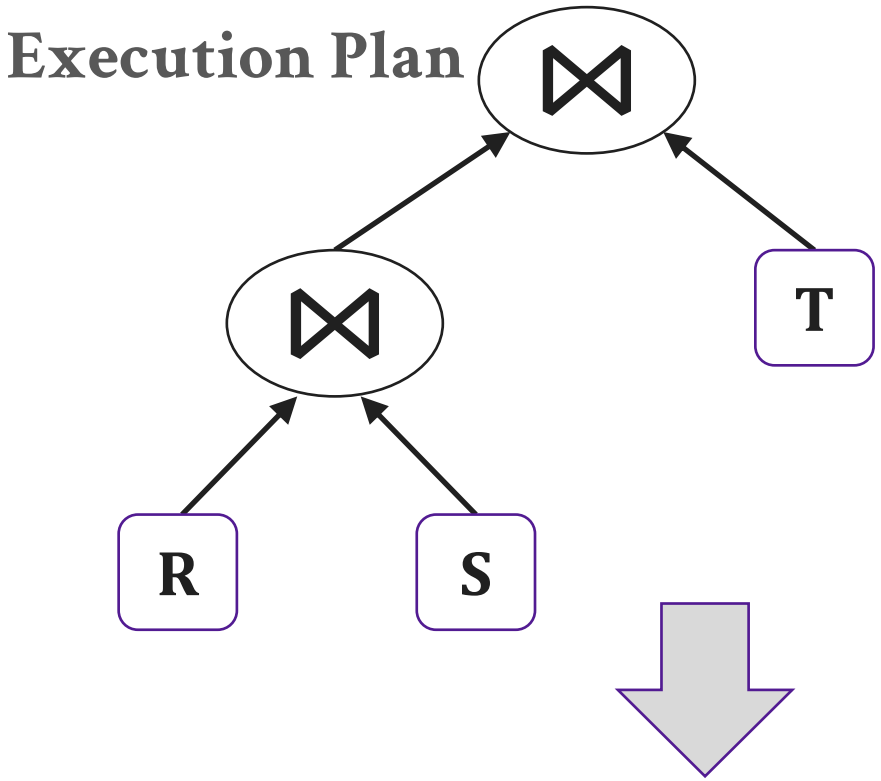


Join Graph (usually a hypergraph)

Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

THE MEMO STRUCTURE

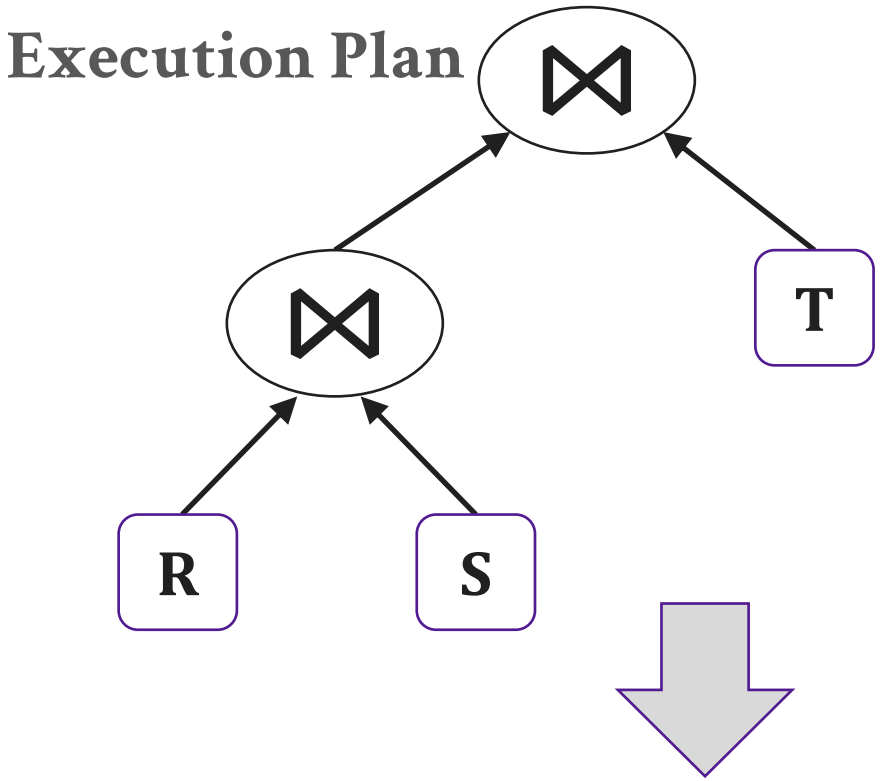


Join Graph (usually a hypergraph)

Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

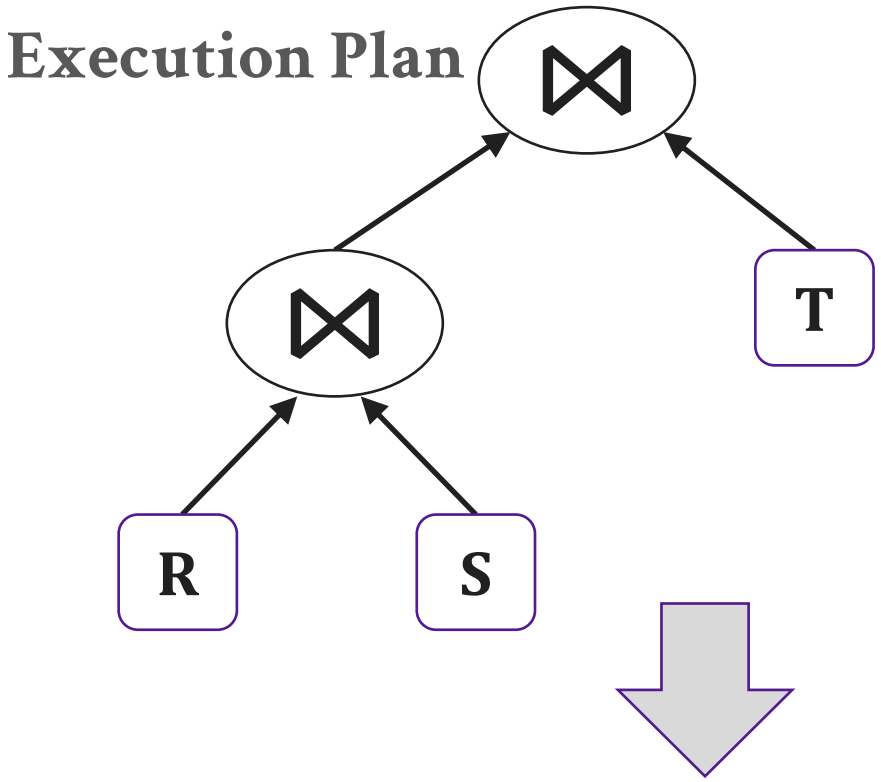
THE MEMO STRUCTURE

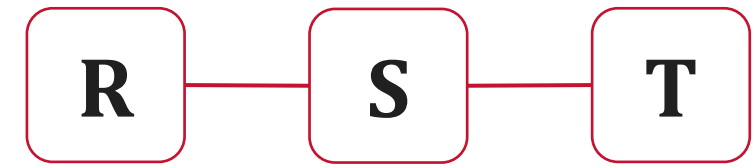


Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

THE MEMO STRUCTURE





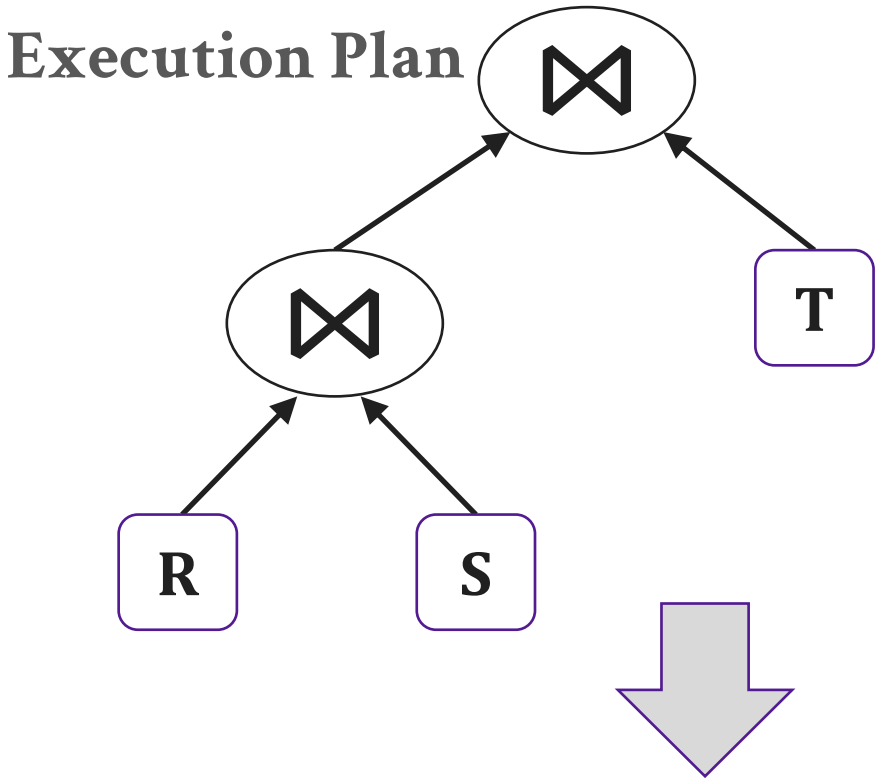
Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

Have a plan at hand now.

Plan 1: (R ⋈ S) ⋈ T

THE MEMO STRUCTURE



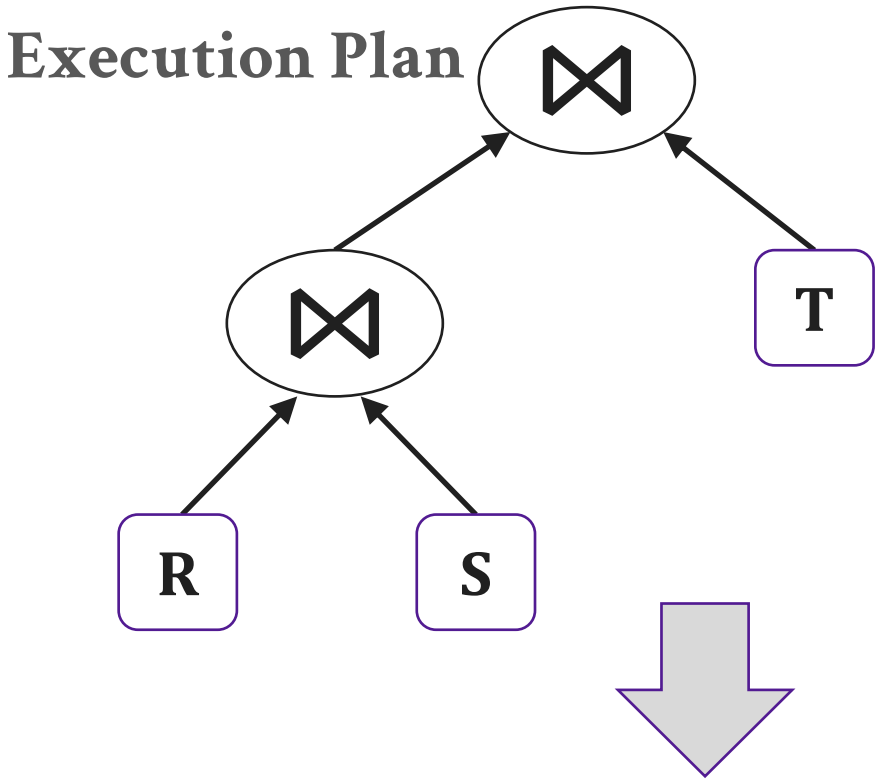
Join Graph (usually a hypergraph)

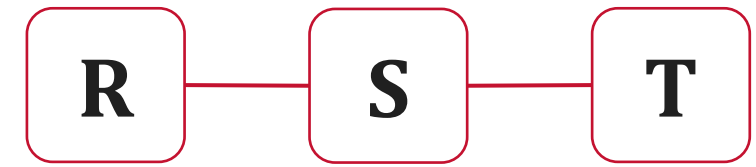
Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

Plan 1: (R ⋈ S) ⋈ T

THE MEMO STRUCTURE



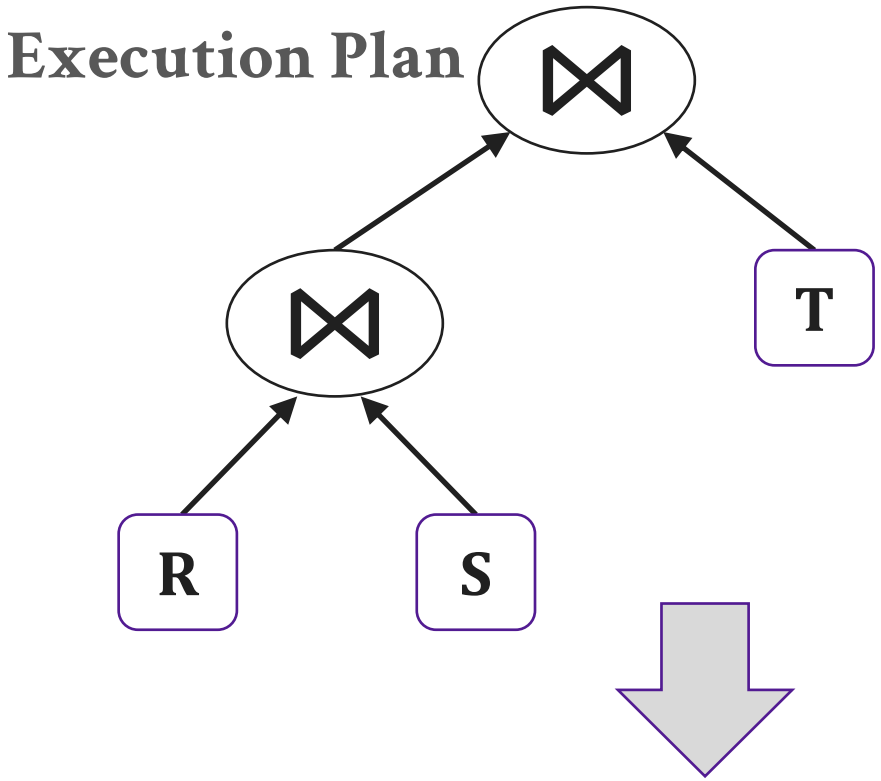


Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R ← Already have this part (in a "memo")
{S}	S
{T}	T

Plan 1: (R ⋈ S) ⋈ T

THE MEMO STRUCTURE

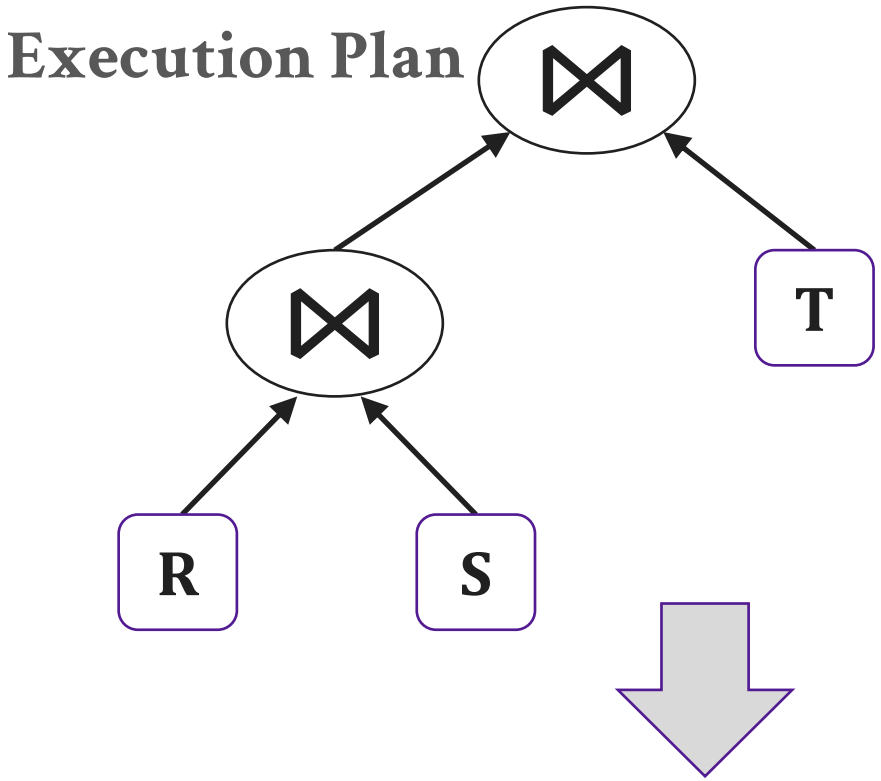


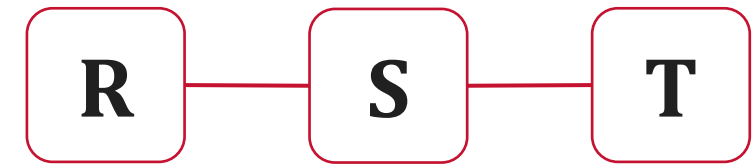
Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

Plan 1: (R ⋈ S) ⋈ T

THE MEMO STRUCTURE



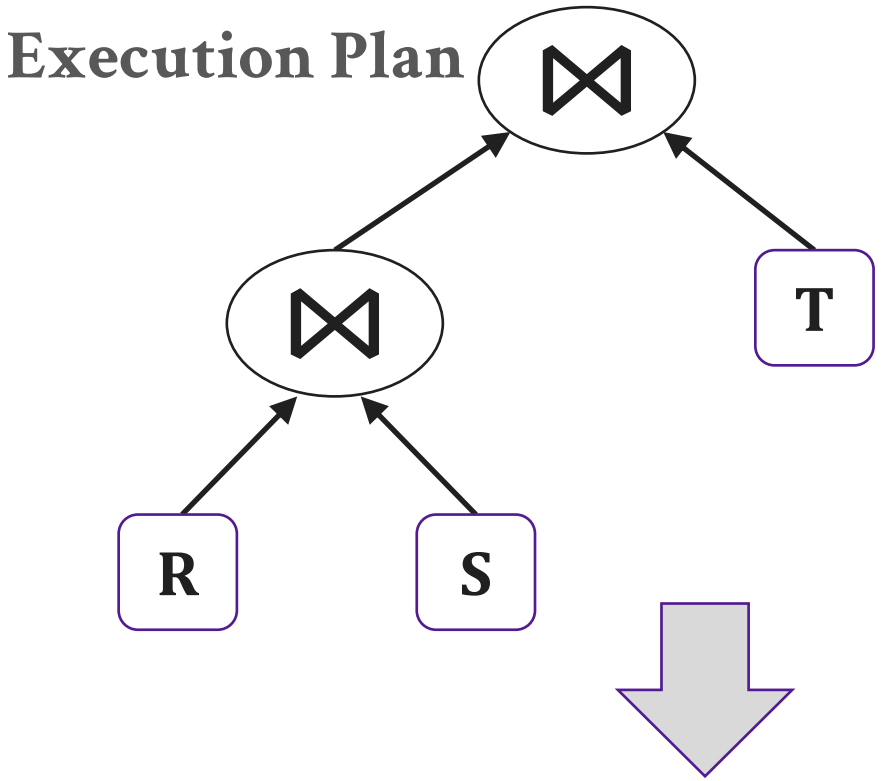


Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S ← Already have this part (in a "memo")
{T}	T

Plan 1: (R ⋈ S) ⋈ T

THE MEMO STRUCTURE



Join Graph (usually a hypergraph)

Conceptual map of what we need to "memorize."

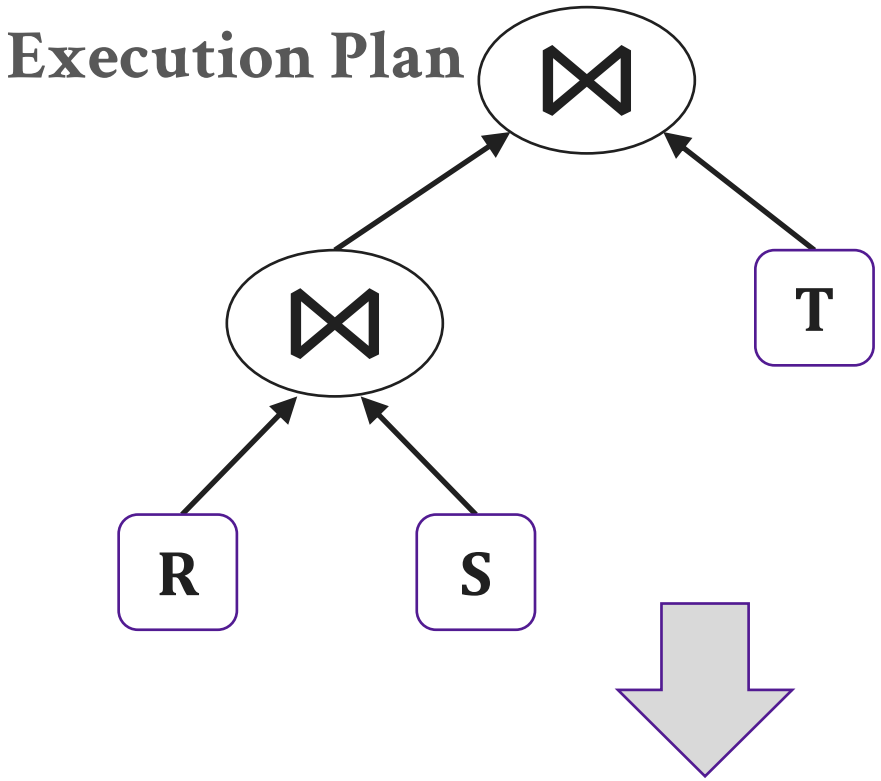
Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

Have a second plan now.

Plan 1: (R ⋈ S) ⋈ T

Plan 2: R ⋈ (S ⋈ T)

THE MEMO STRUCTURE



Join Graph (usually a hypergraph)

Note: Here the search did not proceed in a pure top-down, depth-first order (for that see Figure 2.4 in the paper).

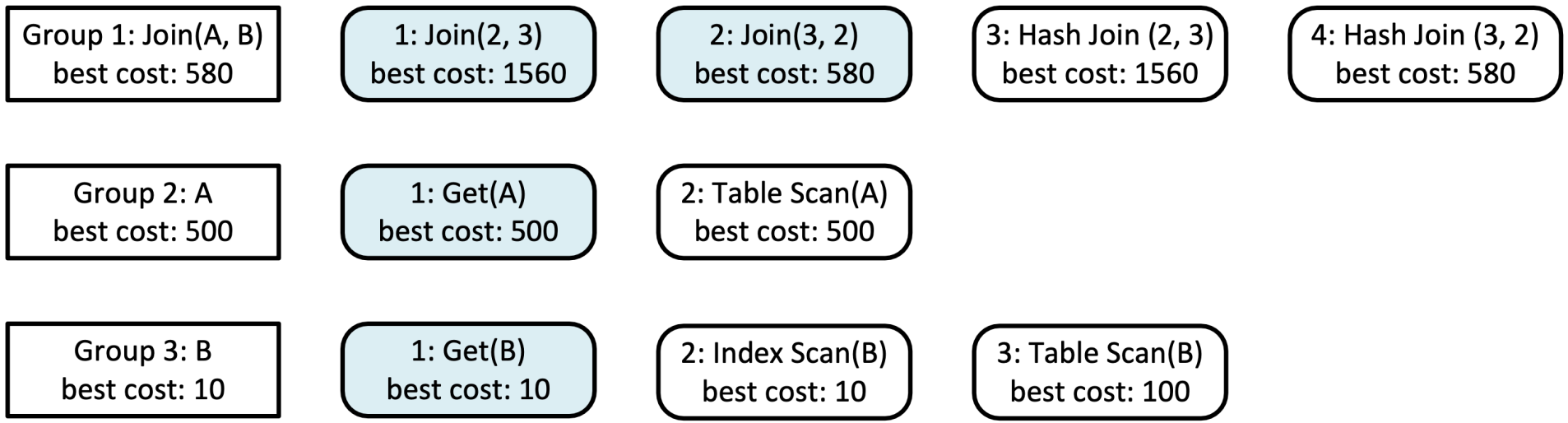
Conceptual map of what we need to "memorize."

Group	Subgroups
{R, S, T}	{R, S}⋈T, T⋈{R, S}, {R, T}⋈S, S⋈{R, T}, {S, T}⋈R, R⋈{S, T}
{R, S}	{R}⋈{S}, {S}⋈{R}
{R, T}	{R}⋈{T}, {T}⋈{R}
{S, T}	{S}⋈{T}, {T}⋈{S}
{R}	R
{S}	S
{T}	T

Plan 1: (R ⋈ S) ⋈ T Plan 2: R ⋈ (S ⋈ T) •••

THE MEMO STRUCTURE

Figure 2.3: An example memo of $A \bowtie B$, where an index is available on table B. Logical expressions are colored in blue. Physical expressions and groups are annotated with the cost of the corresponding best plans.



VOLCANO/CASCADES QO: CORE CONCEPTS

- Idea: "Optimizer generator" → The optimizer reads a set/sequence of rules and explores plans based on matching the rules.
  - Adding a new rule is easy. The core optimizer enumeration method remains the same.
- Rule types:
  - **Logical transformation** rules: Equivalent SQL/RA expressions, e.g., join associativity.
  - **Implementation**: Logical to physical operator mapping; e.g., Inner Join -> Hash Join.
- Notes:
  - An expression (next slide) may be partly in logical form and partly in a physical form.
  - Still have to pick some enumeration order, and need cost-based optimization.

VOLCANO/CASCADES QO: CORE CONCEPTS

- **Expressions**: Logical or Physical
  - e.g.: R⋈S is a logical expr., and HashJoin(TableScan(R), TableScan(S)) is a physical expr.
  - A logical expression can have many corresponding physical expressions.
- An expression has two types of **properties**: Logical and Physical
  - Logical property: what is true of the expression regardless of the physical expression; e.g., output cardinality.
  - Physical property: properties associated with the physical implementation; e.g., sort order, number of threads used to execute the expression, …
- Enforcers: a class of physical operators that enforce physical properties, e.g. sortedness or degree of parallelism.

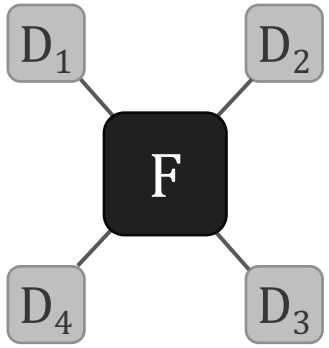
Volcano/Cascades QO: Core Concepts

- Note: Still have to pick some enumeration order, and need cost-based optimization.
  - Still need good histograms/sketches and cost estimation methods.

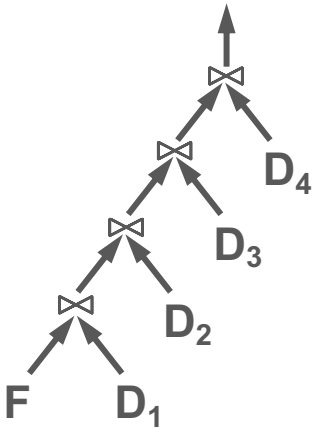
CASCADES QO: CORE CONCEPTS

- Mix generation & cost estimation phases.
  - Find a "good" physical plan without exhaustive exploring the logical search space.
- Improved guidance.
  - Use heuristics to reduce the search space for common join graph shapes to known "efficient" plans.
  - Deactivate specific rules.
- Switch from recursive calling to a non-recursive, task-based style.
  - Avoid program stack overflow, which is capped by the OS to a fraction of the DRAM space.
  - More control over the search direction;
    e.g., derive the best plan of a parent expression after its inputs' best plans are derived.

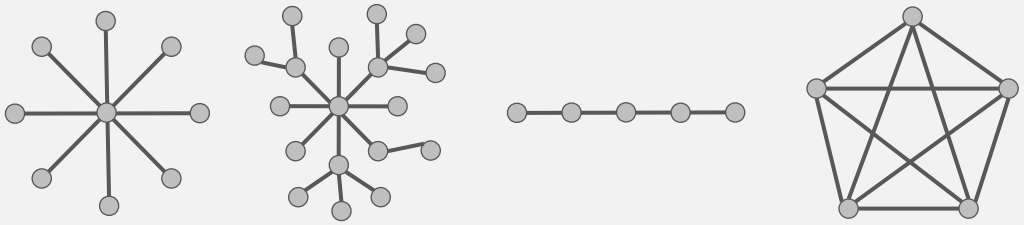
Schema



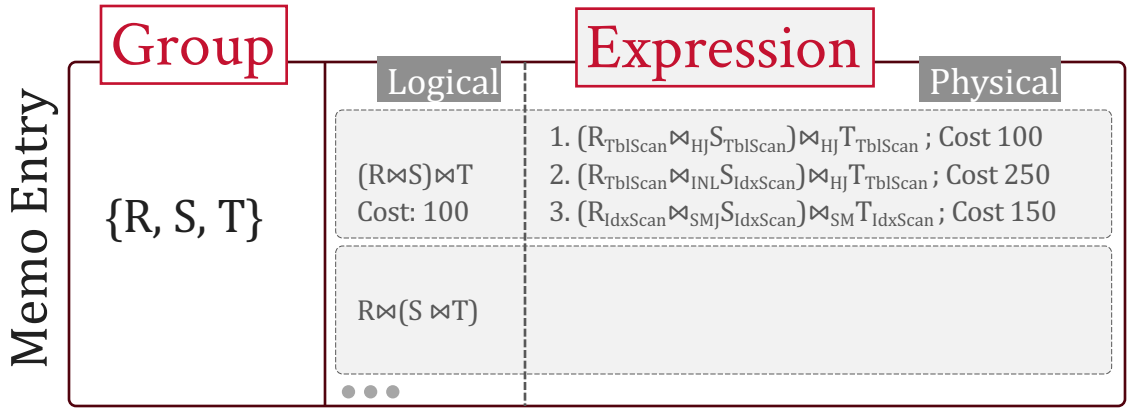
Left-deep plan



Common Join Graph Shapes



CASCADES QO: TASKS

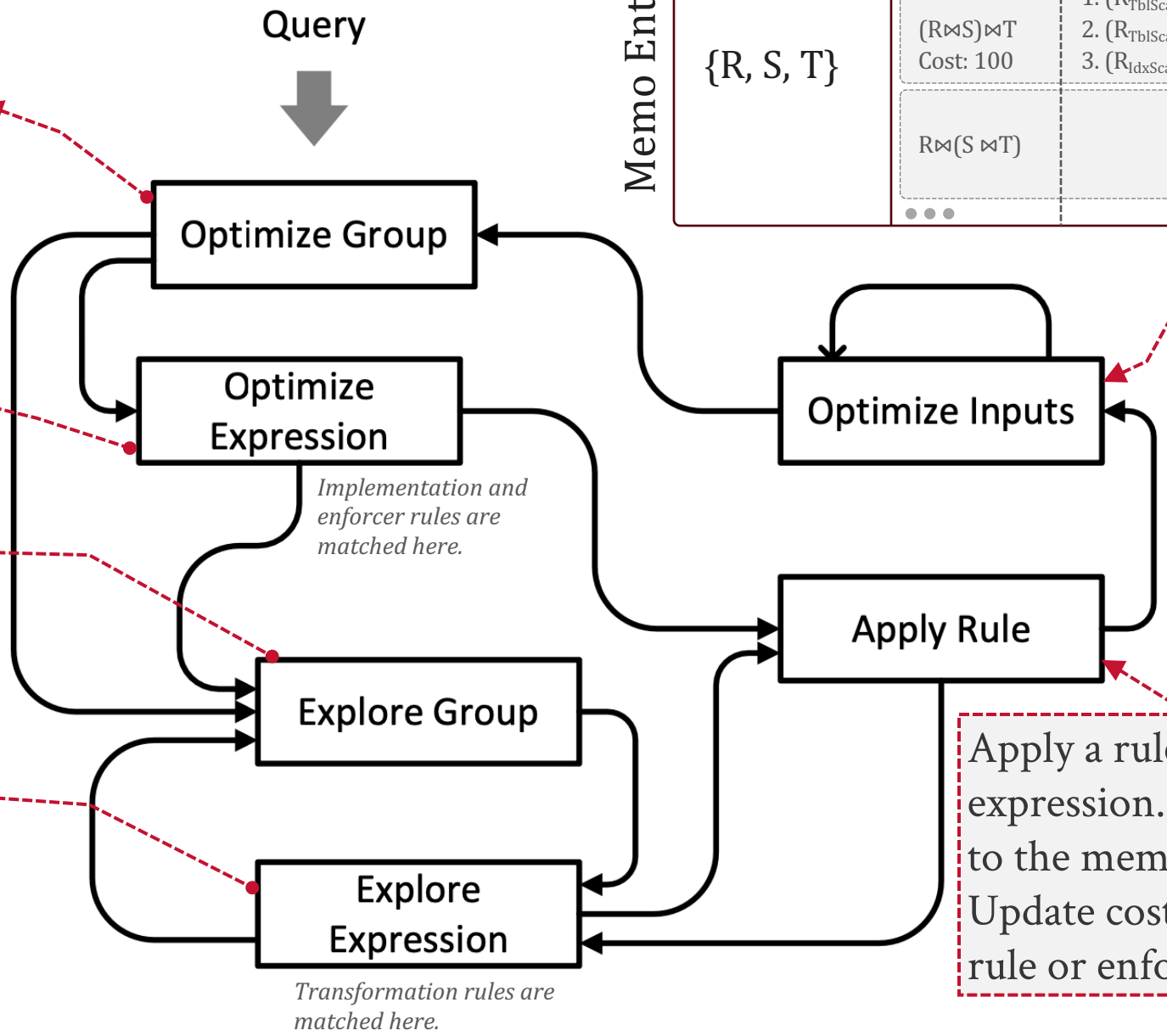


Generate the best physical plan for the group.

Generate the best physical plan for an expression. Use guidance to constrain the generated "moves."

Generate logical expressions for a group.

Generate logical transformations for an expression (follow the promise order); use guidance to skip some rules.



Generate the best physical plan for an expression. Update memo with best plans, including visited sub-expressions.

Apply a rule to an input expression. Add new expressions to the memo, sorted by promise. Update cost if an implementation rule or enforcer is applied.

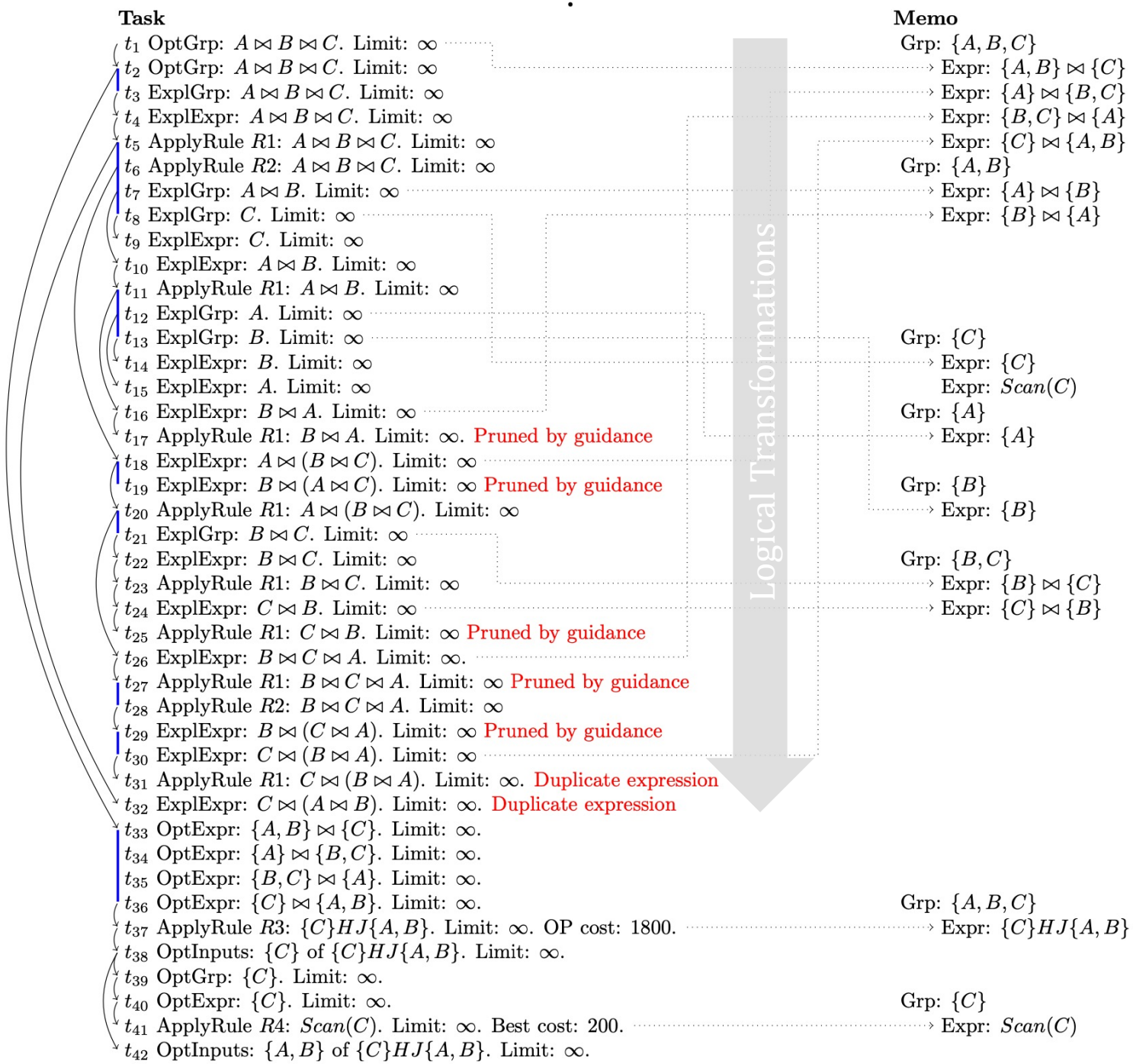
CASCADES QO: EXAMPLE

		Rule	Promise	Guidance
Transformation	**R1**	Join Commutativity	1	No consecutive application
	R2	Join Right Associativity	2	No cross product
Implementation	**R3**	Join to Hash Join	3	N/A
	R4	Get to Table Scan	4	N/A

Expression	Cardinality
A	100
B	1000
C	200
A⋈B	800
A⋈C	20,000
A⋈B⋈C	400

Cross product here, but we may have a better estimate using histograms or sketches.

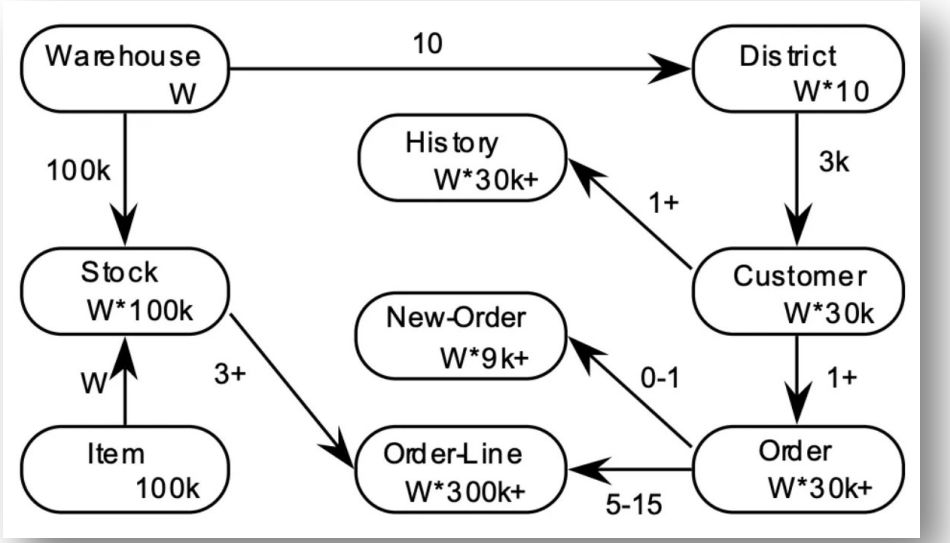
Physical Op	Cost function
HashJoin (X, Y)	$3 \cdot \|X\| + \|Y\| + \|X \bowtie Y\|$
TableScan(X)	$\|X\|$



CASCADES: MULTI-STAGE OPTIMIZATION

- There may be a large number of rules over time.
  - QO time may be too large even for simple queries.
  - OLTP databases are heavily indexed. Common paths are effectively "hot-wired," and the optimal plan is easy to find.
  - OLTP queries are often "simple."
- Optimize in stages, and only do the full QO in the last stage.

TPC-C schema

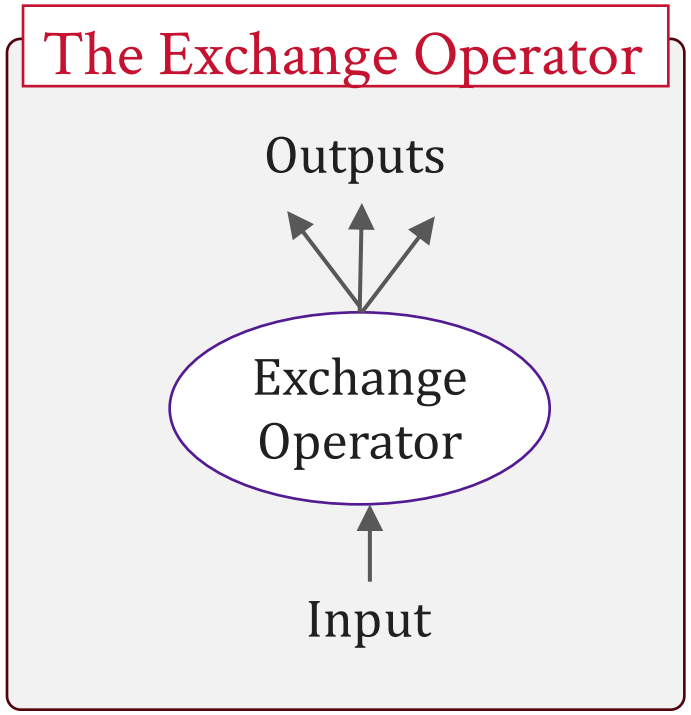


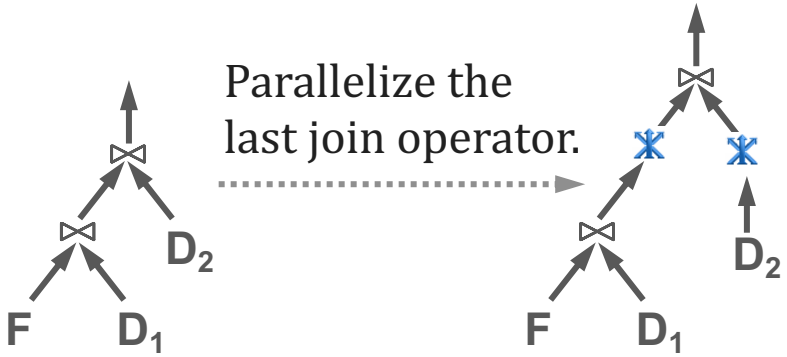
TPC-C query (part of a bigger query)

```
UPDATE customer
    SET c_balance = c_balance - :h_amount,
        c_ytd_payment = c_ytd_payment + :h_amount,
        c_payment_cnt = c_payment_cnt + 1,
        c_data = CASE
                     WHEN LENGTH(c_data) > 500
                     THEN SUBSTRING(c_data FROM 1 FOR 500)
                     ELSE c_data
                 END
    WHERE c_id = :c_id
      AND c_d_id = :c_d_id
      AND c_w_id = :c_w_id
```

Cascades: Other Considerations

- Parallelize an individual search.
  - The task-oriented approach provides a framework, but have to track task dependencies and coordinate.
  - ORCA (Cascades QO from Greenplum) does this. But, there are many open problems related to efficiency.
  - Recall processor parallelism (# cores) is growing and is projected to continue to grow in the future.
- Parallel query operators.
  - Exchange operator.
  - Use enforcers to change the degree of parallelism.





Transformation-based Query Optimization Summary

- A different approach to QO v/s the Selinger bottom-up approach.
  - Based on transformation.
  - It also uses dynamic programming.
  - With a task queue and guidance to shape the exploration of the search space.
- Still need good cost estimation.
- Dealing with nested queries requires special transformation rules, but these are easier to add in a transformation-based QO.
- Many open problems, including efficiency, extensibility, and explainability.
  - Also, can we rethink where the optimization happens? Can it happen during query execution?