iAD: Query Optimization in MARS

Alex Brasetvik
Hans Olav Norheim

Master of Science in Computer Science
Submission date: July 2009
Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Øystein Torbjørnsen, FAST

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Research and get an overview of the state of the art within the query optimization field, both historical and present. Evaluate different approaches, and based on this, research, design and implement (in C#) a query optimizer that suits the needs of Fast's next generation search platform, MARS. Important points include optimization and cost modeling for MARS' expressive algebra, DAG-structured query plans and recognizing common sub-expressions.

The expected result is a documented optimizer prototype that can optimize queries with the most common operators, generating DAG-structured evaluation plans where applicable. It is to be integrated with MARS, but should also be able to run by itself for testing.

Assignment given: 29. January 2009
Supervisor: Svein Erik Bratsberg, IDI

Abstract

This document is the report for the authors' joint effort in researching and designing a query optimizer for *fast*'s next-generation search platform, known as MARS. The work was done during our master's thesis at the Department of Computer and Information Science at the Norwegian University of Science and Technology, spring 2009.

MARS does not currently employ any form of query optimizer, but does have a parser and a runtime system. The report therefore focuses on the core query optimizing aspects, like plan generation and optimizer design. First, we give an introduction to query optimizers and selected problems. Then, we describe previous and ongoing efforts regarding query optimizers, before shifting focus to our own design and results.

MARS supports *DAG-structured query plans* for more efficient execution, which means that the optimizer must do so too. This turned out to be a greater task than what it might seem like — since we must use algorithms that greatly differ from the optimizers we were familiar with. The optimizer also needed to be extensible, including the ability to deal with future query operators, as well as supporting arbitrary cost models.

During the course of the master's thesis, we have laid out the design of an optimizer that satisfies these goals. The optimizer is able recognize common subexpressions and construct *DAGs* from non-DAG inputs. *Extensibility* is solved by loose coupling between optimizer components. *Rules* are used to model operators, and the *cost model* is a separate, customizable component. We have also implemented a prototype that demonstrates that the design actually works.

The optimizer itself is designed as separate component, not tied up to MARS. We have been able to inject it into the MARS query pipeline and run queries end-to-end with optimization enabled, improving the query evaluation time. For now, the project depends on MARS assemblies, but reusing it for another engine and algebra is entirely feasible.

Foreword

Both of us have worked with databases for quite some time — Alex primarily with PostgreSQL, and Hans Olav with Microsoft SQL Server. We both consider ourselves to be well above average interested in databases and their inner workings, including query optimization. However, until now, neither of us have had the opportunity to delve deeply into it.

During the course of the thesis we have learned a lot, and at the same time produced something we believe is a good foundation for *fast*'s further work with a query optimizer for MARS.

We would like to thank Svein Erik Bratsberg and Svein-Olaf Hvasshovd at IDI, NTNU and Øystein Thorbjørnsen at *fast* for their helpful and insightful advice.

We would also like to thank Thomas Neumann at the Max-Planck-Institut für Informatik for his helpful answers about the principles presented in his dissertation. Thanks to Guido Moerkotte at the University of Mannheim for providing us the latest draft of his query optimization book, as well as Red Gate Software for providing us with a free licence for their ANTS Profiler.

Alex Brasetvik

Hans Olav Norheim

Trondheim, July 2009

Contents

1 Introduction **1**
1.1 Goals of The Thesis . . . 2
1.2 How We Have Approached the Problem . . . 3
1.3 Abstract View of an Optimizer . . . 3
1.4 Runtime System . . . 4
1.5 Overview of MARS . . . 5
1.6 Selected Problems Related to Query Optimization . . . 8
1.7 DAG-Structured Query Evaluation Plans . . . 15
1.8 Overview of the Report . . . 19

2 Case Studies and Previous Work **21**
2.1 Introduction . . . 21
2.2 The Early Years: System R . . . 21
2.3 PostgreSQL and Other Open Source Query Optimizers . . . 23
2.4 Rule-based Optimization . . . 24
2.5 Transformative vs. Constructive Optimizers . . . 24
2.6 Reflections . . . 28

3 Design and Implementation **31**
3.1 Introduction and Goals . . . 31
3.2 The Big Picture . . . 32
3.3 Node Structure . . . 35
3.4 Operator Dependency- and Equivalence Mapping . . . 35
3.5 Pre- and Post-Processing . . . 37
3.6 Plan Generation . . . 38
3.7 Share Equivalence and DAG Construction . . . 51
3.8 Graph Pattern Matching . . . 56
3.9 Compound Rules . . . 58
3.10 Integration with MARS . . . 59

4 Cost Estimation and Statistics **61**
4.1 Introduction . . . 61
4.2 Cost Factors . . . 62
4.3 Importance of Statistics . . . 63
4.4 Cost Component . . . 65

4.5 Cost Models for MARS' Operators 66
4.6 Cost Model Implementation 69
4.7 Example Cost Calculation 74

5 Rules: Search Space and Pre-/Post Processing 77
5.1 Introduction 77
5.2 Transformation Rules 78
5.3 Constructive Rules 82

6 Orderings and Groupings 93
6.1 Introduction 93
6.2 Overview 94
6.3 State Machine Model and Implementation 98

7 Example Optimizations 109
7.1 Walkthrough Query 109
7.2 Optimization Steps 109
7.3 The Effect of the Optimization 117
7.4 Additional Optimizations 122

8 Current State 131
8.1 Results 131
8.2 Performance 131
8.3 Identified Issues and Suggested Solutions 135
8.4 Extensibility 137
8.5 Important Missing Features and Implementation Ideas 137

9 Further Work and Conclusion 145
9.1 Further Work 145
9.2 Evaluation and Conclusion 148
9.3 Contributions 149

A Runtime Output 151

B Code Samples 155
B.1 dot Language Sample 155
B.2 Rule Binder Initialization 155
B.3 Share Equivalence Rewrite 157
B.4 Complete, Unsimplified GeneratePlans 157
B.5 BitSet 159
B.6 Optimizer Tests 162
B.7 AbstractTraverser 163
B.8 LogicalJoinTransformer 164
B.9 Lookup Rule 165
B.10 Selection Rule 172
B.11 OrderManager's PrepareOrders 179

C Digital Appendix 183

Bibliography 185

1 Introduction

"Life is what happens while you're busy making other plans."

— John Lennon

Databases and search engines are accessed by executing queries. Ever since the introduction of the automated query optimizer in System R [SAC+79], query optimization has been the subject of much research. Query optimizers are key to enabling user-friendly declarative query languages. Users declare *what* they want from the database — *how* to do it in an efficient manner is then left as an exercise for the optimizer component in the database system. With earlier database systems, such as CODASYL and IMS, users had to program exactly the steps the database had to perform in order to return the desired results. System R and INGRES proved that query optimizers could compete with all but the best programmers [SH05b].

Query optimizers are also often referred to as "*query planners*"[1]. The term "planner" captures another important point of declarative query languages: The way a query is executed can be changed by the system at query time, transparent to the user. For example, as time goes, some tables may be partitioned and/or changed into views. Since the queries are declarative, such changes will not (necessarily) cause old queries to stop working. Thus, they are not solely useful for *optimization*-tasks.

Query optimization is the process of translating an input query to a data structure that is efficiently executable by the system's executor — a query evaluation plan. Query evaluation plans are further described in Section 1.3 and 1.4. In short, a query evaluation plan is a combination of operators that are actually executable by the evaluation system.

For example

$$\sigma_{\mathsf{foo}<42 \wedge \mathsf{A.id}=\mathsf{B.id}} (\mathsf{A} \times \mathsf{B}) \tag{1.1}$$

$$(\sigma_{\mathsf{foo}<42} (\mathsf{A})) \bowtie_{\mathsf{A.id}=\mathsf{B.id}} (\sigma_{\mathsf{foo}<42} (\mathsf{B})) \tag{1.2}$$

are semantically equivalent, but as is, Query 1.2 is probably executed more efficiently than Query 1.1. Query 1.2 can complete in milliseconds, whereas Query 1.1, by making a Cartesian product, can be infeasible to execute.

Optimization is a difficult problem to tackle. The search space grows exponentially not only with respect to relations and their join orderings, but also when different aspects such as recursion, parallelization, distribution, rank-awareness, custom operators, materialized views, multiple query-optimization, etc. need to be taken into consideration.

Furthermore, many specifics related to query optimization are treated as corporate secrets — the better the optimizer, the better the product would perform on benchmarks compared to rivaling products.

This master's thesis is the continuation of our specialization project [BN08], as covered in Section 1.2.

[1] Throughout this report, we'll consistently refer to them as "*optimizers*".

1.1 Goals of The Thesis

We first describe the goals of the specialization project this master thesis continues, and then how they relate to the goals of the thesis. Prioritized, the goals for the specialization project were as follows:

1. Get a broad overview of ongoing efforts within the query optimization research field.
2. Analyze the various approaches and techniques and justify their suitability for a future query optimizer for MARS.
3. Devise a skeleton architecture and design for an optimizer that is clean and extendable, as well as a foundation to implement the techniques found in the previous point.
4. Implement small parts of the architecture and some simple optimization rules. The implementation should lay the foundations for the work in the upcoming master's thesis, and not be so simple it needs to be replaced completely.

We met those goals.

Consequently, the goals of the master's thesis are to further extend our work — i.e. create an optimizer that is able to optimize real queries, and that can exploit some of the unique features in MARS, such as DAG-structured query evaluation plans. It is not realistic that we will have a full-fledged optimizer running for MARS within the course of the master's thesis, though, so the focus is on the most important concepts and not on completeness.

Prioritized, our goals for this master's thesis are as follows:

1. Extend the optimizer with new rules to have it support a wider set of MARS' algebra, at least lookups, joins, sorting and grouping.
2. Implement native support for DAG-structured query plans and multi-queries. The optimizer should be able to construct DAG-ified output plans for tree-structured input plans, if optimal — and optimize several queries simultaneously to get the global optimum.
3. Implement support for tracking available orderings and groupings in a query plan. This is needed to reason about for instance merge joins and streaming groups.
4. Integrate the optimizer with MARS by injecting it into MARS' query pipeline, enabling end-to-end execution of queries with the optimizer.

Implementing support for all the MARS-operators is not important in itself, but taking a rich algebra and showing that our optimizer can deal with it, is.

The reason why hooking into MARS has been given low priority is due to the many difficulties we have had with the provided binaries. MARS is under continuous development, and we have met needs (and provided opportunities) that the developers have not originally thought of. Additionally, we have only been provided with binaries and not source code, so whenever we could not find a workaround for the problems, we have relied on *fast* to fix them for us.

1.2 How We Have Approached the Problem

This section is a brief overview of how we have approached the problem, by describing what choices and priorities we made and when.

We started the work during our specialization project in 2008. Thus, this master's thesis — both the code and this report — is a continuation of that work. We read a large selection of articles regarding query processing and -optimization. We studied the differences between static- and rule-based optimizers — and constructional vs. transformational. Then, we looked at actual implementations of optimizers, first and foremost in the Open Source RDBMS PostgreSQL — both to see an implementation and to get an overview of various transformations, details that are not covered very much in the literature. The details are found in Chapter 2.

We also read a lot of articles about various novel optimization techniques, some of which are briefly mentioned in Section 1.6. We were aware that there was no way we would be able to implement a fraction of what we were reading, but having an overview of future possibilities are important when judging the design and architecture.

With quite an overview of what is out there of optimization techniques and convinced that a rule-based optimizer was the way to go, we still did not know how to solve the problems DAG-structured queries introduce. Then, we found Dr. Thomas Neumann's PhD-thesis about the optimization and evaluation of DAG-structured query graphs [Neu05]. It convinced us we could not reuse much of existing optimization frameworks, so we abandoned the implementation studies and focused our attention solely on the various approaches' ability to deal with query-DAGs. We did not find a lot of frameworks that could handle DAGs , as detailed in Section 2.5.2, so we studied Neumann's framework in depth to figure if it was sufficient for MARS' core qualities, and how we would approach the implementation.

Convinced that Neumann's framework was the way to go, we analyzed how to best adapt it to MARS — with respect to available operators, evaluation and architecture. Then we implemented a basic optimizer based on Neumann's work. We implemented rules for a few operators (i.e. selects, joins and simple scans), and by Christmas we had a rudimentary query optimizer up and running. It did not handle many operators, nor did it deal with orderings, groupings or finding possibilities for sharing within the queries. However, we had the framework up and running, and a good idea of how to implement support for the rest during this master's thesis — and we met the goals of the project.

During this master's thesis, we have continued the implementation. The focus has been on the implementation and adaption of a real (and quite rich) algebra that MARS provides — and not so much on theory, since we covered most ground during the specialization project.

We had some hopes of being able to completely optimize the query shown in Figure 1.2. However, we eventually realized that the general concepts needed to do *all* the possible optimizations of that query would require more work than we had time to, in addition to requiring some features not present in MARS — as commented in Section 1.6.10. As we detail in Chapter 7 and 8, however, we have overcome the most important hurdles, and we have some clear ideas of what would have to be done to complete the optimization of that query. We achieved the most important optimization, i.e. optimizing it into a DAG, as explained thoroughly in Chapter 7.

1.3 Abstract View of an Optimizer

Figure 1.1 shows a simple overview of how a query moves through various steps during execution. Throughout the report, we will use the term **execution** for the entire process from parsing to results are delivered, and **evaluation** for the step where the query plan is evaluated

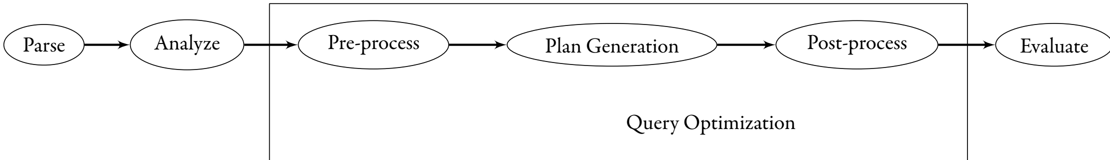


Figure 1.1: Overview of query processing

to produce result sets.

First, the textual query string is parsed into some kind of query graph. Normally, this query graph is a tree, but DAGs are also applicable query graph structures, as discussed in Section 1.7.

Then, the query is analyzed for semantic validity — such as checking well-formedness, that the mentioned relations exist, the user has access, etc.

The *input* to the query optimizer is a query graph that expresses the logical meaning of the query. The query is then rewritten in various ways during the pre-processing phase: Views and sub-selects are flattened, stored procedures inlined, etc. Then, the optimizer decides join orderings and -algorithms, pushes and pulls predicates around, and applies other transformations to ensure as efficient evaluation as possible. A second rewrite phase may be employed, now working on the physical algebra generated in the optimize step.

The output from the final optimization phase is an executable "*plan*", which is a query graph with physical, executable operators. I.e. instead of a logical join operator, a physical operator such as a hash join is used — and instead of logical scans, an index- or sequential scan is used.

When the optimizer considers different equivalent plans, it uses a *cost model*. The cost model defines what "costs" the optimizer should minimize — such as I/O (sequential vs. random), CPU-time, memory, response time and communication costs. To get an idea of what operators cost, the optimizer consults statistics about relations and their data distributions. Cost models and statistics are discussed in Chapter 4.

1.4 Runtime System

The output of the query optimizer is a **query evaluation plan** (hence "*query plan*" or QEP for brevity), a data flow graph which is executed by the query evaluator. This section briefly describes how typical **tree-structured query plans** are evaluated. In Section 1.7.2 we describe how DAG-structured queries differ, and how they can be evaluated. Their complexity is due to the fact that the output of an operator can be the input to more than one operator — thus, the operators cannot simply forget about tuples as soon as they have outputted them, as its consumers can pull with different rates.

A query evaluation plan[2] is a data flow graph where the nodes are **physical operators**. The nodes typically have an **iterator-interface** — i.e. evaluation is done by consecutively calling *next()* on the root node of the tree. Consequently, the root node calls *next()* on its input operators, which in turn calls *next()* on their input operators recursively until a leaf node is reached. For example, if the query is `SELECT ... FROM a JOIN b ON(a.id = b.id)`, the QEP that is evaluated might be a merge join with two clustered[3] sequential scans as inputs. A *next()*-call on the join operator causes it to pull up records from both its inputs until a record that satisfies the join predicate can be produced, which it then emits to the operator pulling from it.

[2]The term "plan" will be used a lot throughout this report. In later chapters, it may *also* refer to lightweight data structures used in the search phase of the optimization — which are not directly executable without translation. What meaning is referred to should be obvious in their respective contexts.

[3]A relation is "clustered" when the logical ordering of the records equal the on-disk physical ordering

Data typically originate from the leaf nodes, which are usually some kind of *scan*-node. Additionally, some nodes might also have a *skip()*-function or have *next()* take an argument that specifies the next desired record. This is useful for example when a merge join operator is joining two clustered inputs, as ineligible records can be skipped instead of attempted joined.

Some operators can be **pipelined**. That is, they output continuously, without needing to process the entire input first. For example, a join operator need not see the entire input before passing tuples joined *so far* to its output node. On the other hand, e.g. sort operators obviously need to see the entire unsorted input before it can output anything. When operators can be pipelined, the overhead of materialization is avoided.

Materialization involves buffering in a temporary relation, which may outgrow permitted or available memory usage and thus need to be flushed to disk. Pipelining is typically preferred, as it decreases needed buffer space and increases alacrity. However, as we will see in Section 1.7.2, when output of one operator can be the input of multiple other operators, pipelining becomes difficult.

1.5 Overview of MARS

1.5.1 Introduction

MARS is *fast*'s next generation search engine. It is a hybrid of a relational database and a search engine. It is designed for information retrieval usage and not transaction processing, but retains many RDBMS-concepts and operators, such as JOINs. *fast*'s existing search engine, ESP®, lacks the JOIN-operator, so the schemas (describing "*documents*") tend to be very denormalized and can thus be costly to maintain and alter — and to query, if the amount of data per document is a lot more than needed by the average queries. In MARS, data is structured into records, contained in *indexes*. The indexes do not have any metadata about relations and foreign keys — referential integrity is not enforced.

These indexes can be joined, either via merge- or hash-joins. Nested loop joins are currently unavailable. Thus only equi-joins are available, so we have focused on those and not full support for theta-joins. A fuzzy-equi-join operator is also available, but we have not looked into it or what it does.

One important goal of MARS is that custom operators should be easy to implement and reuse. Thus, it is very *extensible*.

MARS is written in C# 3.0 on the .NET Framework, which means that we have written our optimizer in C# as well.

However, even though this thesis is mostly about MARS, we frequently use examples from RDBMS-es and SQL, as reader proficiency with those are assumed.

1.5.2 Key Differences

In MARS, query graphs are expressed as directed acyclic graphs (DAGs), and not simply as trees, as is prevalent in most implementations and literature about query optimizers. This allows the output of one operator to be the input of more than one operator, introducing many optimization opportunities — and -problems. See Section 1.7 for more on optimization of DAGs.

Also, MARS is first and foremost a search engine and certainly not a general purpose OLTP- or OLAP-database. It is designed and optimized for queries that will yield results in a short amount of time — search engine users usually expect short response times.

Updates to the index are typically done in *batches*, as updates are expensive because changes in one object may ripple to other objects, e.g. due to relevancy calculations.

Often, one search query should return more than one result set, e.g. grouped by different columns. Therefore, MARS is designed to support simultaneous execution of multiple queries — so the query optimizer must support it as well.

1.5.3 Important Operators

The amount of operators in MARS number in the hundreds, if all operators for their various projects are counted. Clearly, opting to implement an optimizer that handles all of those is infeasible. We have just focused on the most important ones, and briefly describe their characteristics. They will be covered in more detail, but we mention them here as they are referred to throughout the report.

Output is always the root of the query. Every child represent a separate result set. If there is more than one child, the query is a multi-query.

Copy passes the input to multiple outputs, possibly buffering to disk if the readers read at very different data rates. It is the only operator we deal with that can have multiple outputs.

Lookup takes an index name and a word and performs a lookup of the word in the index. The output depends on the index's schema. Typically, indexes are clustered on `DocumentId`, so outputs from LookupOperators are often sorted on it. When the index is a full text index, it is also clustered on the position of every hit for every document.

MergeJoin mergejoins the inputs. There must be at least two input relations, but there is no (practical) upper limit. Inputs needs to be sorted on the *prefix* first attributes of the inputs.

HybridHashJoin hash joins the two inputs on the specified join keys. Applies some optimization if one of the inputs are sorted.

Select filters the input and only lets through the tuples satisfying the specified predicate.

Map is used to perform arbitrary modifications of the input fields, such as renaming, removing and/or applying custom functions.

Group groups the input on the specified group attributes and also performs various aggregates functions, such as count, sum, avg, and so on. May perform streaming or hash grouping, depending on the available ordering.

Near and **ONear** (Ordered Near) filters the input stream of word occurrences, letting only the ones with a distance below a certain threshold through. For ONear, the occurrences have to be in the same order as well, which is used for phrase searches.

ScoreOccurrences applies various scoring functions to the incoming occurrences. Output is grouped on the document id, with scores added.

Trim reduces the result amount to an upper limit, optionally with an offset.

Sort sorts the input on the specified sort fields, optionally reordering the sorted columns to be the first of the tuple — which is required for e.g. MergeJoin and ScoreOccurrences. Additionally, it can perform trimming as well.

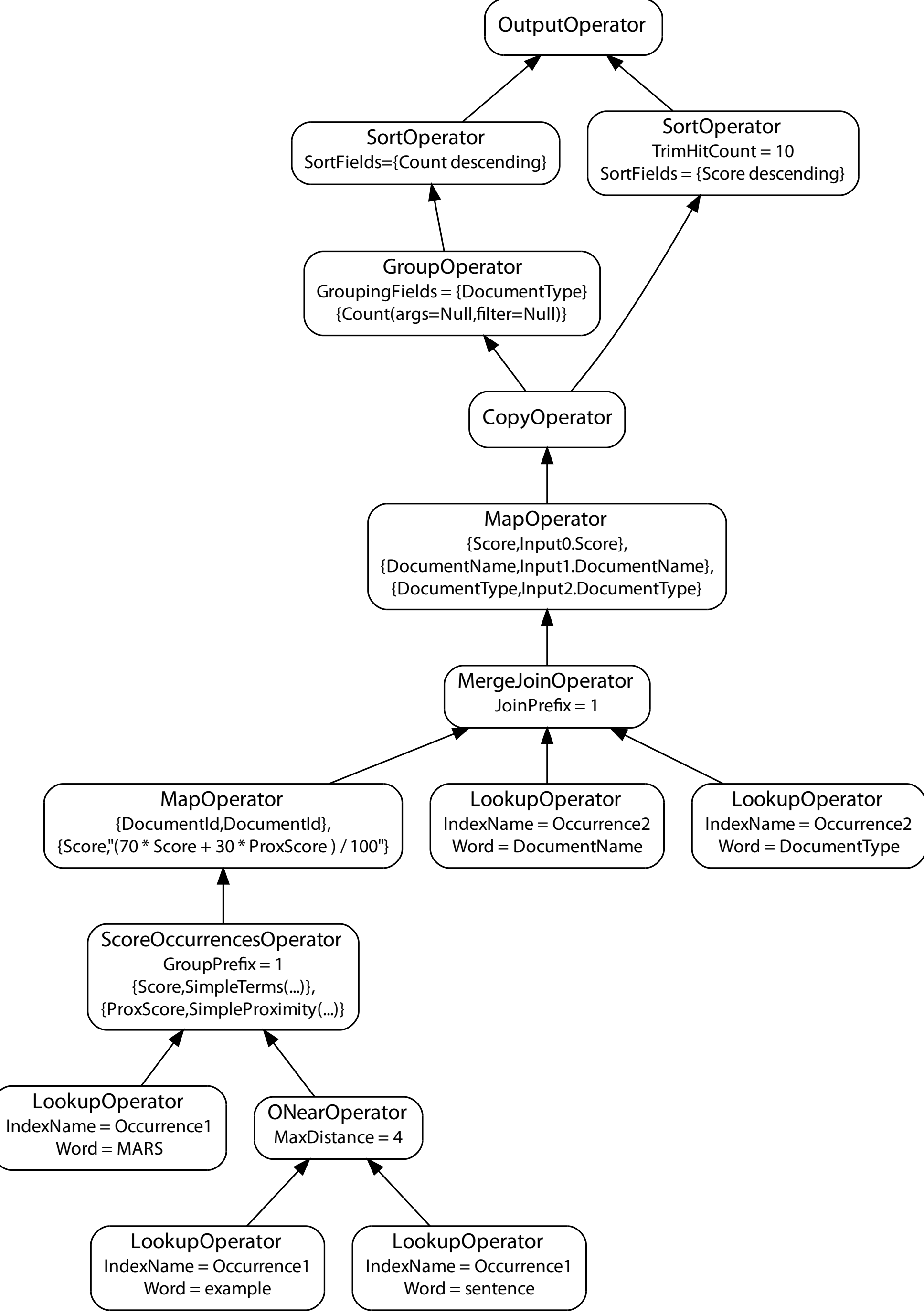


Figure 1.2: Example MARS-query

These are the operators we have chosen to focus on. The list is by no means exhaustive, but the operators are sufficient to express a wide range of queries.

Figure 1.2 shows an example multi-query that searches for documents containing *"MARS"* and the phrase *"example sentence"* and returns the 10 most "relevant" (as defined by the calculations of ScoreOccurrences) results with the name of the documents. Also, the numbers of hits per document type are returned. We have omitted a lot of operator attributes to make the graph fit on a page.

1.5.4 Current State of Query Optimization in MARS

Currently, MARS does not employ a query optimizer. The queries are input to the runtime system as the physical query graph that will be executed. All operators (at least as far as we know) in MARS today are physical operators. For instance, there are HybridHashJoin- and MergeJoin operators, but no Join operator, which is the logical equivalent. Some of them are both logical and physical, e.g. Select and Map.

Moreover, MARS does not currently store any statistics usable for query optimization. This poses a problem when the cost model needs to guesstimate the selectivity of various operators. How we simulated it to work around it is detailed in Section 4.3.2.

1.6 Selected Problems Related to Query Optimization

In this section, we give a brief overview of selected issues related to query optimization. The space of various possibilities of optimizations is so vast, it is infeasible to cover all aspects in a single optimizer — at least while also keeping it extensible and performant. Also, we have a fairly limited amount of time and resources, so we need to restrict the scope of our thesis.

However, it is important to be aware of practical ways of getting better query plans. We want to design an extensible and maintainable query optimization framework that allows new rules and transformations to be developed later on.

Each issue is not equally important — some are just mentioned, while others are covered in depth in other chapters.

Except for the issues that are general for most kinds of query optimization, we shortly reflect on the problem's relevancy to MARS.

1.6.1 Plan Enumeration

Query optimization is a combinatorial search problem. Enumerating all "interesting" plans is expensive — in fact, it is NP-complete [IK84]. The search space is vast and infeasible to explore exhaustively, so we need to constrain it. When doing so, we should prune the bad plans while keeping the optimal plan(s). However, we are doing so without knowing which one that is, using a merely approximate cost model. In reality, we often need to settle for a "good" plan, which is not necessarily optimal, but at least not awful!

Non-exhaustive strategies are either deterministic or probabilistic [LPK+94]. *Deterministic* planners, such as System R's, use heuristics to limit the search space. For example, it only considers *left-deep* join trees, and not *bushy* ones — as shown in Figure 1.3. The bushy plan can be the optimal one, but it is not even considered. *Probabilistic* optimizers, such as *Simulated Annealing* and *Iterative Improvement*, randomly choose a query plan or transform the query according to some probability.

Left-deep plans are plans where all joins have a base table as its right input, and thereby any other subjoins as its left. If we only consider plans without cross products, the size of the

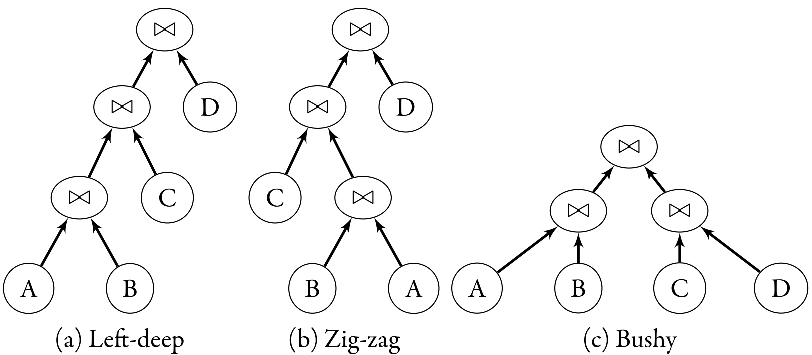


Figure 1.3: Three query trees

search space for left-deep plans for n relations is 2^{n-1} [Moe06]. Left-deep plans are also easily pipelined, as described in Section 1.4

Zig-zag plans are plans where all joins have at least one base table as input (left or right), and without cross joins, the size of the search space is 2^{2n-3} [Moe06].

Bushy plans have no restrictions on join inputs, and give a search space of size $2^{n-1}\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ is the Catalan Numbers, which grow in the order of $\Theta\left(4^n/n^{3/2}\right)$ [Moe06]. Bushy plans are more amenable to parallelization. Consider Figure 1.3c. It is clear that the computation of C ⋈ D need not wait for the completion of A ⋈ B.

Which enumeration strategy our optimizer uses can be configured per optimization. However, it is not clever enough to figure out which strategy to use by itself.

1.6.2 Operator and Predicate Migration

By migrating certain operators and predicates, we can often achieve more efficient plans. For example, we often want to push selects through joins, as selects can be cheaper than joins. Consider the query[4] $\sigma_{\text{person.id}=42}(\text{person} \bowtie \text{city})$, that is selecting a particular person from the join of all people and all cities. Clearly, $\sigma_{\text{person.id}=42}(\text{person}) \bowtie \text{city}$, that is selecting a particular person and *then* joining with city, is much more efficient.

However, this is not always true. For example, assume we have an index on person.city_id and consider a query for all people born after 1950-01-01 living in Å[5]:

$$\sigma_{\text{person.birth}>1950-01-01}(\text{person} \bowtie \sigma_{\text{city.name}=\text{Å}}(\text{city})) \tag{1.3}$$

If we assume the database holds the population of Norway, the amount of people living in Å is certainly less than those born after 1950-01-01 anywhere in the country. Thus, doing a selection on person before the join in this case can be more expensive. Instead, we want to use the index on the join key, and *then* apply the person.birth-predicate. Even if we had an index on person.birth, the predicate would not be *selective* enough to justify index lookups. To determine this, the optimizer needs *statistics* that suggest the *distribution* of the values. See Section 1.6.4.

Another interesting case is when predicates are user-defined functions, which can be expensive to execute. These are discussed in Section 1.6.7.

[4]In our examples, we value clarity over design best practices.

[5]A small village in the municipality Moskenes, Lofoten, Norway

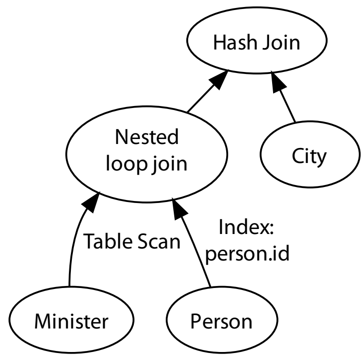


Figure 1.4: Example of a physical evaluation plan

1.6.3 Access Path Selection and Join Ordering

An *access path* is a specific way to access the records. It can be a full table scan (also called sequential scan, file scan, clustered index scan in various systems), or one of several available indexes. In the previous section, a query accessed a person-table which had indexes on its primary key as well as person.birth. Both these indexes as well as a full table scan could be considered when performing the query — with different costs. Access path selection is the determination of which access method is the better one. It also involves considering properties of the returned results as well, such as *ordering*.

When joining several tables, there is usually several *orders* in which they can be joined. Consider the tables person, city and minister, where the latter holds information about the government. We want to display information about all ministers, including information about their home city, that is city ⋈ person ⋈ minister. In what order should the joins be performed? Since the number of ministers is a lot less than the entire population, it is clear that joining city ⋈ person first is suboptimal, because of low selectivity. In fact, city and minister are probably both so small their sizes are negligible. It is person we need to avoid costly access paths on. A reasonable join order, then, is joining minister and person first: city ⋈ (person ⋈ minister). The physical plan could look like Figure 1.4.

1.6.4 Statistics Maintenance and Cost Estimation

When deciding what access paths to use and in which way to order joins, the planner needs to consider relation- and join cardinalities and the relations' value distributions. In the previous example, we reasoned that a certain join ordering was a good one, due to the sizes of the input relations. That is a statistic that the planner needs access to. If this statistic is outdated or otherwise *wrong*, it may cause the planner to choose horrible plans.

Typical information stored about relations, is their cardinality, size in pages, etc. These provide information about the cost of a full table scan. However, we often also want statistics about the value distribution of certain columns. For example, in Section 1.6.2 we reasoned about the distribution of people based on their age/birth date. By doing so, we can reason about a predicate's *selectivity*. Doing so is important when weighing the cost of different access paths.

These statistics are costly to maintain. It is infeasible to provide accurate statistics about value distributions, so they are instead *sampled*. Moreover, storing these statistics in an efficient

and accurate manner is also a concern.

Another valuable use of statistics is to reason about data *correlations*. Such knowledge can be valuable when evaluating access paths and join orderings.

Usage and maintenance of statistics are discussed further in Chapter 4.

1.6.5 Partitioning, Parallelization, Replication and Distribution

When dealing with large data sets, or data sets that are rarely coupled, it is reasonable to partition the data. How the data is partitioned clearly affects how it is queried. For example, if data is partitioned on several nodes in a round-robin fashion, it is likely that every node must be queried in order to get all relevant results. However, data can also be partitioned on ranges and arbitrary (mutually exclusive) constraints, a technique which also makes sense to employ on single nodes. For example, with a constraint ensuring that only "recent" (for some definition of recent) data reside in a partition (and older or archived data residing in any number of other partitions), a query optimizer can ensure that the excluded partitions are not searched, which can greatly reduce the evaluation costs. This technique is called **constraint exclusion**.

Query execution can also often be **parallelized** — both with multiple CPUs and/or with **multiple nodes**. With cheaper and more powerful commodity hardware, this is becoming an increasingly interesting avenue [GHK92].

When done right, this will certainly speed up the query execution, but it also introduces new problems. Dependencies in execution are clearly important, but communication costs complicate the cost evaluation: not only do we need to consider lots of different plans, but we also need to consider *where* sub-plans are executed, what data they have *locally*, and predict the costs of *transferring* results from one node to another.

MARS has support for an Exchange-operator, which is used to handle data exchange when parallelizing execution, but our *fast*-representative told us to focus on the basic issues on *one node* first.

1.6.6 Heterogeneous Environments

Distributing and parallelizing execution on numerous nodes become even harder when the environment is composed of several different application stacks. If there are multiple ways of executing the query, it can be difficult to reason about the costs of the partial problems that can be executed on different nodes. As with parallel execution, mentioned in the previous subsection, we may also need to consider the communication costs in the cost model.

Although interesting for MARS, for example by integrating with SQL Server, this is an even harder problem than those of the previous subsection.

1.6.7 User Defined Functions

An important property of several database systems is their *extensibility*. Users can develop custom functions that are executed *by* the database system.

Such functions can appear both as values, in predicates and as relations:

- `SELECT id, frobnicate(value) FROM …`
- `SELECT … FROM … WHERE … AND coverage(…) < 42`
- `SELECT … FROM generate_series(0,1000)`

Consider the following example of the second case, where we have a user defined function as a predicate, due to [HS93]:

```
/* Find all channel 4 maps from weeks starting in June that show more than 1%
snow cover. Information about each week is kept in the weeks table, requiring
a join */
SELECT maps.NAME
FROM maps JOIN weeks ON (maps.week=weeks.number)
WHERE weeks.month='June' AND maps.channel=4 AND coverage(maps.picture) > 1
```

In this case, *coverage(...)* is an expensive user defined function. If we naïvely *push* all predicates below joins, we will be calling *coverage(...)* on a lot more rows than if we *pull* it *up* and apply the restriction *after* the join.

In addition to considering whether the user defined functions are expensive we also need to consider whether they are *volatile*, *stable* or *immutable* [Pos08a]:

- **Volatile** functions can do anything — return different results for each invocation, and modify the database. An optimizer cannot optimize its usage: it has to be re-evaluated every time.
- **Stable** functions cannot modify the database and promise to return the same value for the same input arguments *in a single statement*.
- **Immutable** functions are as stable functions, except they will *always* return the same value for the same input arguments.

Only stable and immutable functions can be optimized. But how do we determine their cost? Eventually, we clearly need to provide some interfaces to allow user defined functions to inform the optimizer about their evaluation characteristics. MARS emphasizes that it must be easy to develop custom operators and functions. Hence, these issues are realistic. We cover these issues briefly in Chapter 4 and Section 8.5.3.

1.6.8 Rank-Aware Optimization

Ranking functions define a measure of relevance of an input record. They are often used in a context where we want records within certain *boundaries* of the *score* determined by the ranking function — or the top-k.

[ISA+04] introduces *rank-join*-operators, which progressively rank the join result. As soon as they can be certain the top-k results have been ranked, the operator returns — without spending more time on records that cannot "win".

The authors argue that by enabling efficient evaluation of ranking queries, relational databases can efficiently answer Information Retrieval queries. Hence, these techniques may be interesting in MARS, which is an attempt to combine the best from relational query engines and search engines, as discussed in Section 1.5.

MARS does not currently have rank-join-operators, so we have not studied them further. However, since the architecture must support more than one join operator anyway, we believe that support for a rank-join-operator can be added as another join-helper-rule. These are described further in Section 5.3.9.

1.6.9 Multi-query Optimization

We often need to perform multiple queries to get the result page structure we want. For example, a product search on an online store can produce results grouped by producer, price range, customer reviews, availability, and so on. In this case the results of all the queries are the same, but ordered differently. Clearly, it is not *necessary* to perform all those queries from scratch

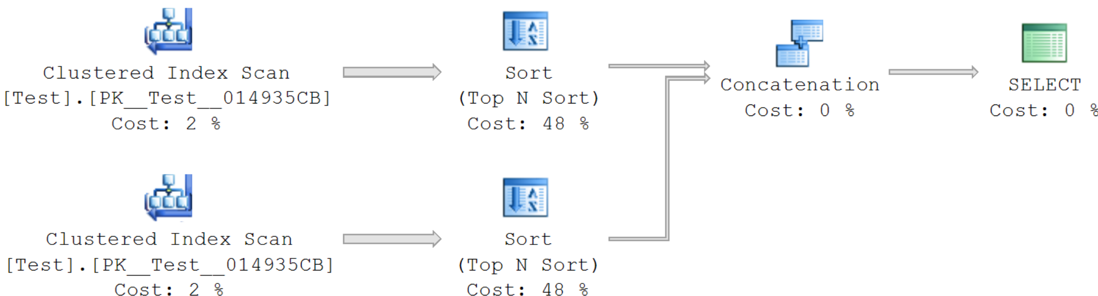


Figure 1.5: Example query tree which could be optimized in a query-DAG (screen shot from Microsoft SQL Server)

for every ordering needed, but few query optimizers consider these possibilities. For example, given the query `SELECT * FROM (SELECT TOP 50 * FROM test ORDER BY bar ASC) t1 UNION ALL SELECT * FROM (SELECT TOP 50 * FROM test ORDER BY bar DESC) t2`, which takes the 50 first and 50 last tuples from test ordered by bar, SQL Server 2008 produces the plan depicted in Figure 1.5. This could have been solved better by using a DAG and not scanning the input relation more than once. In this case, there is no index on test.bar. The "clustered index scan" is in reality a table scan.

In Section 1.7.1, we show a more thorough motivating example regarding multi-query optimization. MARS supports multi-query execution, and they are amenable to being structured as DAGs and allow sharing of intermediate results. Optimizing multi-queries is an important goal.

1.6.10 Inferring Function Semantics

The goal of semantic query optimization is to use application- and/or domain-specific knowledge to optimize queries.

In [CZ98b], Cherniack and Zdonik describe how some rewrite rules are *too general* to be expressed with rewrite rules. For example, transforming arbitrary boolean expressions into conjunctive normal form cannot be expressed with a simple rule. On the contrary, some rules are *too specific* to an application context to be a generic optimization rule. For example[6], consider the following two OQL-queries:

1. `SELECT DISTINCT x.reps.capital FROM x IN S`

   (the capital cities represented by the senators in S)

2. `SELECT DISTINCT (SELECT d.mayor FROM d IN x.reps.cities) FROM x IN S)`

   (the mayors of cities in the states represented by the senators in S)

In these queries, it is possible to skip the duplicate elimination. Because of the *semantics* of the relations, the intermediate results are already free of duplicates: a state only has one senator, a city can just be the capital of one state, and a city only has one mayor. Such semantics are not limited to foreign keys between relations. They develop two languages, "COKO" and "KOLA", that express rewrite, transformations and when they are fired. The rules are also automatically verifiable by a theorem prover.

MARS does not currently have any features regarding inferring function semantics — not even foreign key relationships to suggest how different data relate. We have therefore not studied this any further. However, it could be interesting to some time in the future allow develop-

[6] Example due to [CZ98b]

ers to express semantics about their data and relationships, to better aid the optimizer's decision process.

Interestingly, the lack of foreign- and primary key information prevents an optimization opportunity found in the example query in Figure 1.2 (page 7). In that query, it is advisable to perform the join of `DocumentName` *after* the `SortOperator` which also trims the amount of results to 10. However, to be able to move that join above the trim, the optimizer has to be certain that in doing so, it does not change the amount of tuples — otherwise it would be changing the semantics of the query. If the join key had been a primary key (which it is, but the optimizer cannot tell), it could have inferred that the semantics would have been preserved.

1.6.11 Adaptive Query Optimization and Dynamic Query Plans

An adaptive query processing system is one that considers *and* monitors the state of its environment to determine its behavior [HFC+00].

In large scale database- and search engines, utilizing numerous nodes, failures are inevitable. Thus, it is important to be able to devise good plans, also in the presence of node failures and variable availability, as well as detecting this situation quickly.

To be able to choose good plans, the optimizer needs reliable and accurate statistics, to estimate selectivity and cardinality. Changed statistics immediately affect the decisions of the optimizer, so how do we ensure a high fidelity between the actual data (which is part of the environment) and the statistics? One suggested method is to use results from performed queries to maintain statistics [CR94]. This enables continuously maintained statistics. These issues are discussed a bit more in Chapter 4.

In [CG94], Cole and Graefe describe how "static" query plans, made with assumptions about selectivity and resource availability at compile (optimization) time, can be sub-optimal for their actual (possibly changing) run-time invocations. The environment can even change *while* the query is running! For example, a node can suddenly disappear, as mentioned in the previous paragraph.

They introduce an operator "choose-plan" that is executed run-time to reevaluate the current evaluation plan. For example, if the selectivity estimation of a selection turned out to be estimated wrongly (detected by the evaluation system), the join orderings can be reconsidered. Also, the optimizer can decide certain points in the plan where plan-reconsideration could make sense — perhaps due to *uncertainties* with selectivity estimation (detected by the optimizer).

MARS does not currently even have an optimizer, so the choose-plan-operator is certainly not available. Being able to alter the plan on the fly is also likely to necessitate considerable changes to the runtime-system. Therefore, we have not studied this any further. However, we imagine this could be added as a post-processing step.

1.6.12 Genetic Query Optimization Algorithms

When dealing with *very* complex queries, the search space can get too large even with efficient pruning. For example, the PostgreSQL ORDBMS uses a genetic query optimization algorithm when the number of relations to be joined is $\geq$12 [Pos08a].

A genetic optimization algorithm uses a non-deterministic, random search. Possible plans are considered a population of individuals. Individuals each have chromosomes and genes. By simulating evolutionary processes, such as *mutation* and *selection*, new generations of individuals with better properties than their ancestors are introduced [Moe06, Pos08a].

Genetic algorithms are not simply random guesses for a solutions. The search uses stochastic processes, so it is better than random [Pos08a].

Since query optimization is exponential in nature, and MARS could possibly have to deal with queries that are too complex for our optimizer to handle, genetic optimization algorithms are one possible approach. However, we are not knowledgeable about this subject, and our *fast*-representative has told us not to worry too much about the really complex queries.

1.6.13 Proving correctness

Query optimizers repeatedly transform and change the query. The goal is always to achieve a better plan *without* changing the semantics of the query. However, doing so provably correct becomes increasingly difficult when the number of rules and transformations increases, as the number of possible combinations of the rules explode. This is especially true for extensible query optimizers, where rules and transformations are implemented by plugins and are not a part of the optimizer core. It is easy to test a new rule in isolation, but hard to predict how it interacts with and influences the existing rules. In [CZ98b], Cherniack and Zdonik argue that rules are best expressed *declaratively* and not in *code*, to be able to verify rule correctness automatically with theorem provers. However, they acknowledge that the expressive power of automatically provable rules are not sufficient to express many necessary query transformations.

Generally, proving correctness approaches the unfeasible when complexity increases. Thus, pursuing the *provable* is not *practical*. To remedy this, the optimizer must be easily *testable* by design. Every rule and every component must be testable in isolation — and in combination. We have a few unit tests that assert the outcome of the optimization, but since we have not implemented too many rules yet, testing infrastructure and -helpers have not been prioritized. However, changing various components to use a *dependency injection*-pattern to ease "mockability" does not require substantial effort.

1.7 DAG-Structured Query Evaluation Plans

As the concept of DAG-structured query evaluation plans (DQEPs) is so central to our work, we need to present it early on.

The most common way to represent query graphs is as *tree structures*. Tree structured query evaluation plans are easier to optimize and execute than DAG-structured plans, but also less flexible. With trees, output of one operator can just be input to *a single parent operator*. Consequently, intermediate results cannot be reused, which is a major limitation.

As we elaborate in Section 2.6, we have based most of our work on a PhD-thesis by Dr. Thomas Neumann. This section explains advantages and difficulties with DAG-queries.

We start with a motivating example. Section 1.7.2 describes some of the challenges with DQEPs, while Section 3.7 discusses *share equivalence* — i.e. the property that partial results can be shared.

1.7.1 Motivation

Using DQEPs introduces many challenges compared to those that are tree-structured, so a motivating example to see why the extra effort is worth it is warranted.

A multi-query is a composition of multiple queries, where every single query returns its own result set. Typically, when multiple queries are executed, they are executed in isolation. Whether they are executed serially or in parallel does not matter, as they are unaware of each other and the partial results of the other queries. Sharing is largely limited to locality in time with respect to page caching.

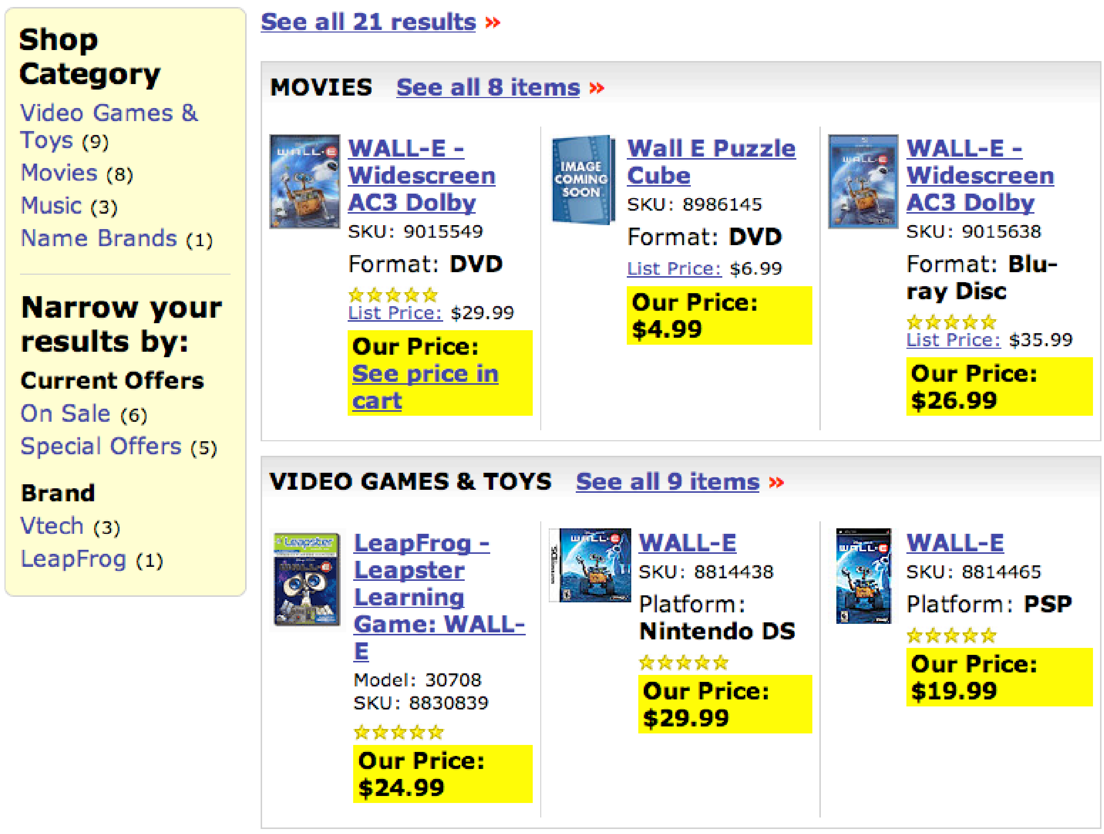


Figure 1.6: Example of search that results in multiple smaller searches

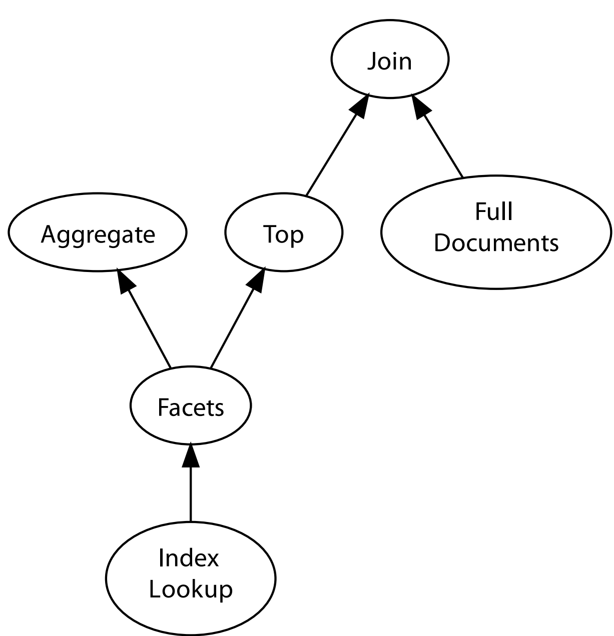


Figure 1.7: Imaginary DAG-structured Query Evaluation Plan

Consider for example the query shown in Figure 1.6. It is a search for "wall-e" on Best-Buy.com, which is backed by *fast*'s ESP®. Knowing that ESP® does not support DAGs, it is *reasonable to believe* that in order to present the results shown in the figure, the query is actually *multiple smaller queries*. We do not know exactly how the search is evaluated, but present an approach that is not unreasonable, and then how it could be performed more efficiently with DQEPs. In the next two paragraphs, we present two example evaluation strategies, where we have taken the liberty to deal with some imaginary operators to keep the example simple. The first example uses simple queries, and the last example uses DQEPs.

First a query for "wall-e" returns a list of all results that has anything to do with "wall-e". The results, which are just item pointers, are used to determine interesting categories (collectively called "facets" (e.g. "Shop Category") and "facet values" (e.g. "Music") in search engine lingo) to show results from — such as video games and movies. Then, for each interesting category, a search is performed to find the three most relevant hits for the input query for that category. This may seem excessive for the "wall-e"-search that only yields 21 results, but searching for "DVD" yields 100319 results — and iterating over all of them just to put the three most relevant results into each category is too expensive.

In Figure 1.7, we show an imaginary DQEP which reuses partial results. First, "wall-e" is looked up in the full text indexes. The result is a list of pointers. These pointers are then used by a "facet"-operator which returns a list of facet values and their respective pointers — for example `music=[1,2,3]`. The result of this operation is then sent to an operator that aggregates the counts of each facet value, *and* to an operator that finds the three most relevant results for each of them. Then, the output is joined with the full result data. This results in two result sets — a list of relevant facets values (with counts), and a list of relevant results for each of them. These two result sets can then be combined to present the result page shown in Figure 1.6.

1.7.2 Challenges

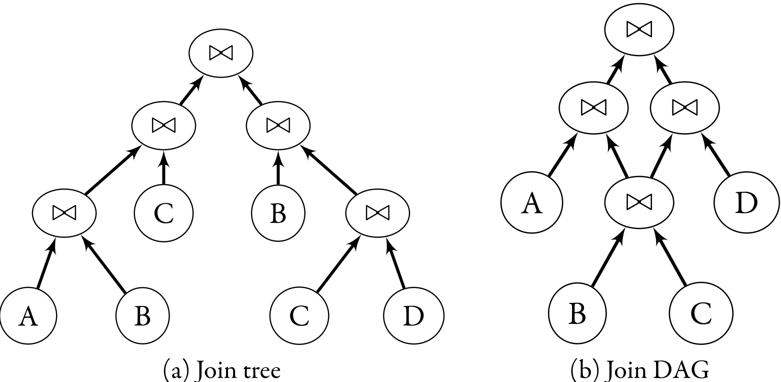
DQEPs are inherently more difficult to deal with than their tree-structured counterparts. In this section, we describe why finding the costs and evaluating the queries are more challenging.

Lack of Optimal Substructure and Cost Estimation

Optimizers dealing with trees typically employ dynamic programming- and memoization-techniques. These rely on an optimal substructure to combine optimal solutions of sub-problems to achieve an optimal solution. While tree-structured QEPs have this property, DAG-structured do not. For example [7], consider the query $A \bowtie B \bowtie C \bowtie B \bowtie C \bowtie D$. Figure 1.8a shows a possible solution where the optimizer first finds that $(A \bowtie B) \bowtie C$ and $B \bowtie (C \bowtie D)$ are partial optimal solutions, which are then combined. If the sub-optimal partial solutions $A \bowtie (B \bowtie C)$ and $(B \bowtie C) \bowtie D$ had been considered, the *common sub-expression* $(B \bowtie C)$ could have been shared, resulting in the plan in Figure 1.8b. Consequently, an optimal DAG cannot be constructed by simply combining optimal partial plans. This is also an example of a non-multi-query where the DAG-structure is beneficial.

The cost of computing intermediate results is only paid once, even though the result is used by several operators. This fact must be captured by the cost model that compares plan alternatives, so that the cost of $(B \bowtie C)$ is only counted once even though it is used by two parent join-operations. Reading it n times is not free either, but at least not n times the cost. Section 4.4 covers the design of a DAG-aware cost component.

[7] Example adapted from [Neu05]



(a) Join tree (b) Join DAG

Figure 1.8: Two equivalent join graphs

```
1 SELECT s.CompanyName, c.CategoryName, COUNT(p.ProductID) AS Count FROM Products p
2 JOIN Suppliers s ON p.SupplierID = s.SupplierID
3 JOIN Categories c ON p.CategoryID = c.CategoryID
4 GROUP BY s.CompanyName, c.CategoryName
5 WITH CUBE;
```

Listing 1.1: Sample query resulting in a shared temporary relation

Runtime System

Evaluating DQEPs is a lot more involved than evaluating tree-QEPs. With trees, the output is just sent to a single operator, so the output can be forgotten as soon as it has been sent. However, with DAGs, multiple operators can receive the output — and with an iterator model, they may not necessarily request the data in an orderly fashion. One operator may even *finish* processing the output before another one has *started* — and the output may not necessarily fit in memory.

This is not really an optimizer issue, and has already been implemented in MARS.

In [Neu05], Neumann identifies four approaches:

1. Transforming the DAGs to trees. This defeats the purpose of having DAGs in the first place.

2. Only share output from materializing operators. This adds no extra overhead, but severely limits the sharing opportunities.

3. Use temporary relations.

4. Push output into operators instead of having them pull.

The third alternative can easily be identified in current commercial implementations. For example, the plan generated by SQL Server 2008 for the query in Listing 1.1 is shown in Figure 1.9. Sharing of the temporary relation is triggered by the `WITH CUBE`-construct, which makes SQL Server produce several combinations of the groups [Cor08]. The figure has been modified to make it fit on a page, and to highlight the two operators that share partial results.

The fourth alternative, pushing output, is what MARS uses, and what Neumann concludes is the most fruitful approach. Its details are outside the scope of this thesis.

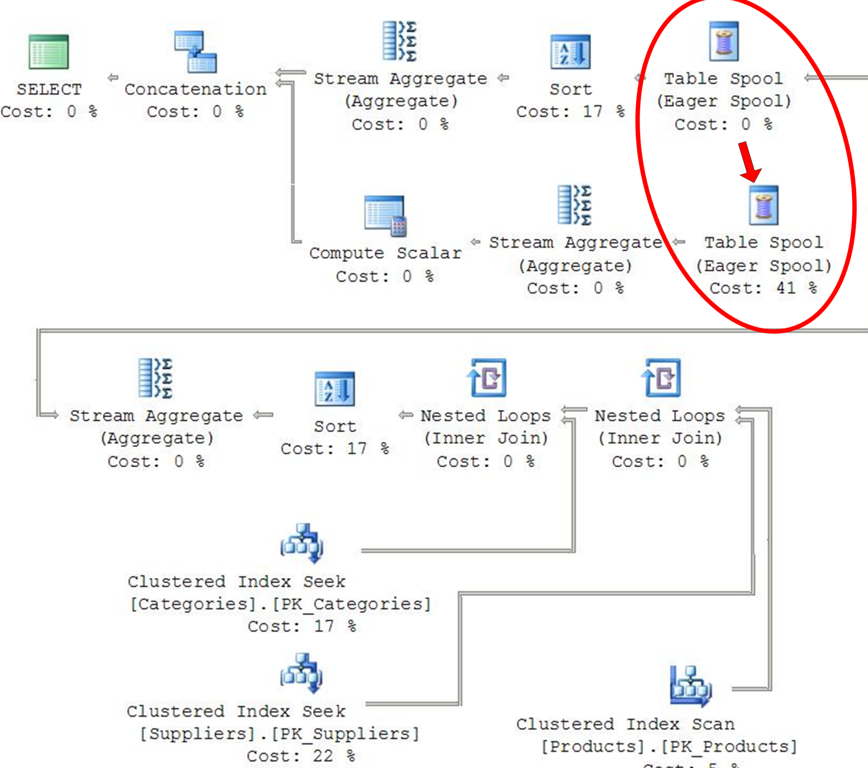


Figure 1.9: Screen shot of a plan generated by SQL Server 2008 sharing a temporary relation

1.8 Overview of the Report

The rest of the report is structured as follows.

Chapter 2 talks about various approaches to query optimization, especially rule-based approaches, and gives an introduction to transformative vs. constructive optimization. We also give a brief description of the System R-optimizer, which is considered a seminal work in the area of query optimization. Chapter 3 is the bulk of the report, and describes our optimizer implementation, both design and algorithms. Chapter 4 gives an introduction to costing of query plans and ends with a description of the cost component in our optimizer.

Chapter 5 presents the rules the optimizer implementation uses. We discuss the rule interface, and describe the implementation of a few rules. Chapter 6 covers the quite involved process of determining orderings and groupings. Chapter 7 contains a complete walkthrough of the optimization of an example query — to put all the components into context. Also, several small example optimizations are covered, to show what the optimizer is capable of.

Chapter 8 discusses the results of the thesis and the current state of the optimizer, while Chapter 9 concludes the report and wraps up suggested future work, and how it may be approached.

Appendix A lists the output of a query execution run with optimization enabled, while Appendix B includes selected code samples, and Appendix C describes the contents of the accompanying digital appendix.

Code Samples

The report includes quite a few code samples to illustrate how the optimizer implementation works. Most are given in C#, which is the language we have used for implementation. Common for all of them is that we have focused on making them easy to read and understand, em-

phasizing the important concepts. This means that we have simplified most of them, removing things not necessary for understanding. Most of the code will therefore not compile as it is. The reader is referenced to the digital appendix for the complete source code.

In our code, we *do* follow C# coding guidelines, but we have sacrificed them in the report to save space.

2 Case Studies and Previous Work

"Any fool can make history, but it takes a genius to write it."

— Oscar Wilde

2.1 Introduction

In this chapter, we describe some systems and articles we have looked into in more detail than those listed in the "Selected Problems"-section in the Introduction.

We start out with System R, a historically important system, which was successful partially due to its query optimizer. Then we describe PostgreSQL, an open source database system with a solid query optimizer. However, most of the chapter is devoted to rule-based and transformative- or constructive approaches to query optimization. We describe what model we settled on for our query optimizer, and — more importantly — *why*.

As explained in Section 1.2, these studies were conducted during the specialization project last semester.

2.2 The Early Years: System R

System R pioneered query optimization, proving that declarative, easy-to-use query languages are a viable means of interfacing database systems. Its design choices have influenced many current relational query optimizers. We mention it due to its historical importance, and to have a coherent and succinct description of a working optimizer.

In the seminal article "Access Path Selection in a Relational Database Management System" [SAC+79], Selinger et al. described the techniques used in System R. It is a bottom-up optimizer which uses a dynamic programming algorithm to find the left-deep plan that minimizes the *cost* of the overall plan. The cost calculation is explained later. Possible scans are sequential scans, and clustered and non-clustered index scans. Hash joins were not available — nested loops- and merge-joins were the two possible join operations. Indexes were implemented as B-trees.

The optimizer begins by parsing the query into *blocks*, which are then optimized one by one. If queries are nested, the nested sub-queries are treated as subroutines which return tuples to the predicates they occur in. Queries are not rewritten to *flatten* sub-queries, however.

For every block, all available access paths for the accessed relations are considered, paying attention to *cost* and *interesting orders* — that is orders compatible with the block's `ORDER BY`- or `GROUP BY`-clauses as well as equi-join predicates. The cheapest plans are kept for further consideration.

What	Selectivity $\mathcal{S}$
`column=value`, with index i	$\frac{1}{\text{ICARD}(i)}$
`column=value`, without index	$\frac{1}{10}$
`NOT predicate` p	$1 - \mathcal{S}(p)$
`column1=column2`, both indexed	$\frac{1}{\max(\text{ICARD}(i_1),\text{ICARD}(i_2))}$
`column1=column2`, one indexed	$\frac{1}{\text{ICARD}(i)}$
`column1=column2`, none indexed	$\frac{1}{10}$
`column IN (list` l `of values)`	$\min\left(\frac{1}{2}, \lvert l\rvert \times \mathcal{S}\ (\texttt{column} = \texttt{value})\right)$
`predicate` p_1 `OR predicate` p_2	$\mathcal{S}(p_1) + \mathcal{S}(p_2) - \mathcal{S}(p_1)\,\mathcal{S}(p_2)$
`predicate` p_1 `AND predicate` p_2	$\mathcal{S}(p_1)\,\mathcal{S}(p_2)$

Table 2.1: Selectivity estimates in System R

Situation	Page Fetches
Sequential scan	TCARD/P
Unique index matching an equal predicate	2[1]
Clustered index i matching at least one predicate	$\mathcal{S}(\text{predicates}) \times (\text{NINDX}(i) + \text{TCARD})$
Non-clustered index i matching at least one predicate	$\mathcal{S}(\text{predicates}) \times (\text{NINDX}(i) + \text{NCARD})$
Same, but small enough to fit in memory	$\mathcal{S}(\text{predicates}) \times (\text{NINDX}(i) + \text{TCARD})$
Clustered index i not matching any predicate	$\text{NINDX}(i) + \text{TCARD}$
Non-clustered index i not matching any predicate	$\text{NINDX}(i) + \text{NCARD}$
Same, but small enough to fit in memory	$\text{NINDX}(i) + \text{TCARD}$

Table 2.2: Estimated number of page fetches in System R

To be able to estimate approximate costs for access paths, *statistics* about the various relations are fetched from the system catalog. The statistics were not maintained continuously, but updated periodically with an `UPDATE STATISTICS`-command. The statistics kept are

- NCARD (t) — the *cardinality* of relation t.
- TCARD (t) — the number of *pages* in the segment that holds tuples of relation t. There can be tuples of other relations in the same segment, thus ...
- P (t) — the fraction of pages in the segment that contains tuples of relation t.
- ICARD (i) — the number of distinct keys in index i.
- NINDX (i) — the number of pages in index i.

Independence between columns is assumed. However, they also implicitly assume a *uniform* distribution of the values — no statistics regarding the *data distribution* are kept. This leads to rather simple selectivity estimates. We list a few them in Table 2.1. The article also lists selectivity estimate formulae for `column > value`; `column BETWEEN value1 AND value2`; and `column IN sub-query`. We omit them for brevity.

These selectivity estimates are key to devising the cost estimates listed in Table 2.2. We list just a few. We set CPU costs $= w\,|\text{RSI calls}|$, where $|\text{RSI calls}|$ is an estimate of the number of tuple-handling instructions, and w is a factor weighing processing costs and I/O. $|\text{RSI calls}|$ is the product of relation cardinalities and the selectivity factors of the involved predicates. Generally, the cost $\mathcal{C} =$ page fetches $+$ CPU costs.

With these cost estimates for single relation scans, the optimizer can search for a cheap join ordering, by considering many possible join trees. To limit the size of the search space to something that is feasible to explore, some heuristics are used: System R limits the search space to left deep join orderings and defers alternatives with Cartesian products as a last resort. It does so by considering the join-predicates linking the various relations together. For example, if we have relation A joined with B, and B joined with C, with predicates that are incompatible — i.e. they do not form a transitive closure — then $(\mathsf{A} \bowtie \mathsf{C}) \bowtie \mathsf{B}$ and $(\mathsf{C} \bowtie \mathsf{A}) \bowtie \mathsf{B}$ are not considered. However, $(\mathsf{A} \bowtie \mathsf{B}) \bowtie \mathsf{C}$ and $(\mathsf{B} \bowtie \mathsf{C}) \bowtie \mathsf{A}$ may have very different costs, so they both need to be considered.

The optimizer uses a dynamic programming algorithm, which relies on the optimal substructure inherent in query *tree* optimization: the optimal solution to $n - 1$ joins is needed to find the optimal solution to n joins. First it finds the cheapest single-relation plans with various interesting orderings. Then, every relation is joined as an *outer* relation with all other relations, giving all possible two-relation plans. This process is repeated n times, where n is the number of relations. For each pass i, the i-th relation is joined as the outer relation to all $(i - 1)$-relation plans. Even though heuristics are employed to prune the search space, the number of plans checked still increases exponentially, with a complexity of $\mathcal{O}\left(n2^{n-1}\right)$.

The most important contribution of the System R optimizer, besides proving that optimizers were a viable alternative to "database programming", is the use of statistics and cost functions, coupled with a dynamic programming algorithm to devise cheap plans.

2.3 PostgreSQL and Other Open Source Query Optimizers

PostgreSQL is the most advanced open source database system. It was started as a research project by Michael Stonebraker, as "Postgres" at the time — a followup to Ingres [Pos08c]. We consider the source code to be of high quality and that it is easy to read, and several articles about key design decisions have been published, such as [SRH86,Sto87]. Thus, it was a natural candidate for studying — a real implementation, with many features and the source readily available.

We also looked into the optimizers of MySQL, SQLite and MonetDB, but chose to focus on PostgreSQL — because it is more feature complete, and the code was a *lot* easier to read than that of MySQL and MonetDB. SQLite is quite lightweight, with sub-query flattening as the most interesting feature. We did not prioritize achieving breadth in the study of *implementations*.

The optimizers in the mentioned products are also static — i.e. they do not have a rule-based architecture. If any of the alternatives had been rule based, they would have been more interesting to study.

Studying PostgreSQL gave us some insight of quite a few optimization transformations that are not typically mentioned in the literature. We list some of them in Section 5.2.3. The study was something we started out with in the initial phase of the specialization project, to have studied a real implementation and to get an overview. However, as we progressed with the literature studies, *we realized that the algorithms and data structures used by PostgreSQL*

[1]Although they do not explicitly mention it, it is clear they assume that the internal nodes of the B-tree indexes reside in memory. Thus, there's one page access to get the pointer in a leaf node, and one to fetch the actual page.

would not directly apply when having to deal with DAG-structured query plans — for reasons detailed in Section 1.7.2. Thus, we abandoned the implementation studies to concentrate on the important differences between DAG- and tree-based optimizers. None of the other systems mentioned deal with DAG-structured query plans either — we are unaware of *any* Open Source DBMS that do, as it is still a novel approach. Hence, it is of little use to go into details about the results of studying PostgreSQL. It was certainly useful to have looked into a real implementation, seeing how a query is handled as it goes from pre-processing to plan-generation to post-processing. Also, since the transformations in PostgreSQL are hard-coded, it was (although not bad per se) a contrast to our rule-based approach.

2.4 Rule-based Optimization

There are generally two kinds of optimization architectures: Some are hard-wired, and some are rule based. To avoid confusion — with "rule based", we do *not* refer to Oracle's "rule based optimizer", as opposed to their "cost based optimizer". Here, rules as objects is a design decision, not the optimizer in its entirety.

Hard-wired optimizers have their transformations and rules hard-coded. The optimizer is then aware of all possible operators and their semantics. Adding new operators and transformations will likely involve rewriting large parts of the optimizer.

Rule based optimizers are *extensible*, with a *modifiable* set of optimization rules [CZ98a]. The architecture allows rules to easily be added, which also enables adding new operators the optimizer was previously unaware of. Rules can even be specified by the user "on the fly" [PHH92]. The optimizer core then knows nothing about actual operators, as it is just orchestrating rule instances and comparing their outputs using a cost model. One of the earliest uses of a rule-based optimizer, was Squirel [SC75], a transformation-based optimizer dating back to 1975. However, the most referred articles on the subject are those of Starburst [PHH92] and the EXODUS-Volcano-Cascades-series of optimizer *generators* [GD87, GM93, Gra95]. We describe these further in Section 2.5, where we also describe the model we base our optimizer on.

There are two types of rules: those used to **pre- and post-process** the query, and those used in the search phase of the optimization. Pushing `NOT`s down as far as possible is an example of a pre-processing rule, and merging selections is an example of post-processing. These rules are fairly simple, and more examples are mentioned in Section 5.2.3. Pre- and post-processing constitute just a fraction of the total optimization time for queries with many relations — most time is spent in the search phase. They are usually not cost modeled.

The rules of the **search phase** determine how the search space is explored and is usually cost modeled. Thus, these obviously need to be treated efficiently. When search rules are developed, one must be careful not to cause the search space to explode unnecessarily.

2.5 Transformative vs. Constructive Optimizers

When searching for better plans, two approaches can be distinguished: transformative and constructive.

Transformative optimizers consecutively *transform* the input query to an *equivalent* and hopefully cheaper output plan. The input and output are always semantically equivalent. This is a nice property, as it enables aborting the optimizer at any time and returning the best plan so far — for example due to a time budget or a hard deadline. We describe a few approaches in Section 2.5.1.

Constructive optimizers take the *logical goal* of the query, and then rebuilds the query from scratch — assembling one block at the time. These can also be classified into top-down and bottom-up. We describe two approaches in Section 2.5.3.

Note that this distinction applies to the *search phase* of the optimization. Even though constructive optimization is done in the search phase, transformative rules are typically applied in the pre- and post-processing phases of the optimizer.

2.5.1 Transformative: EXODUS, Volcano and Cascades

EXODUS, Volcano [GM93] and Cascades [Gra95] are three projects by Goetz Graefe et al., which are successive refinements to a rule-based optimizer *generator*. We describe them in terms of how the successor improves the predecessor.

With **EXODUS**, a database *implementer* defines a *model description*, which contains the list of operators, what methods should be considered when building and comparing access plans, transformation rules, and implementation rules, which map logical and physical operators [GD87]. For example *join* → {*hash join, inner loops, Cartesian product*}. The model is then used to generate C code, which in turn is compiled and linked with the implementer's model to achieve a specific optimizer. Rules are generally described declaratively, but can also be supplemented with C code when necessary. Adding new operators and rules involved changing the model and then generating a new optimizer. The most important contributions of EXODUS were proving that an optimizer *generator* framework could work, based on declarative rules and transformation on logical and physical algebra [Gra95].

However, the authors identified several limitations with EXODUS, and found it difficult to produce efficient, production-quality optimizers [GM93]. This lead to the development of the **Volcano** optimizer generator. The goals of Volcano was to be usable with existing query evaluators and as a separate tool. Additionally, it should be more efficient in terms of optimization time and memory consumption during search space exploration — all while remaining extensible and permitting parallelization, use of heuristics and model semantics to guide the search and to prune bad paths early. As EXODUS, it used a model to generate code, which in turn was linked to the implementer's database system. It also has a separate logical and physical algebra. However, in EXODUS, they were treated with a suboptimal data structure, which resulted in an inability to capture requirements about physical properties (such as ordering), inefficient memory usage and an overhead in *re*analyzing existing plans. For large queries, EXODUS actually spent most of the time reanalyzing existing plans [GM93]. This was solved with a dynamic programming algorithm and memoization in Volcano. Additionally, Volcano had a more flexible cost model, which delegated the comparisons to functions provided by the implementor. The most important contributions of Volcano was improving EXODUS shortcomings with more efficient algorithms and data structures, which in turn enabled more extensibility [Gra95].

Having used the Volcano optimizer generator in two different projects, its authors identified additional design flaws, whose remedies were the goal of the **Cascades**-project. Cascades is not as well-published as the other architectures. We contacted Goetz Graefe and was told [Gra95] is the only article published about Cascades, but we also found mentions of Cascades elsewhere ([Bil], [ONK+95]) which suggested there were more to it. According to its paper, it is the foundation for the optimizers found in Tandem's NonStop SQL and Microsoft's SQL Server. It is no longer an optimizer *generator*, but a framework where rules are provided as objects. Rules are no longer encoded in a formal specification which is subsequently converted, and can even be specified and generated at runtime.

[Gra95] lists several of Cascades advantages compared to Volcano. We highlight a few of them:

- Rules as objects
- Operators that may be both logical and physical
- Patterns that match an entire sub-tree
- Incremental enumeration of the search space
- Optimization tasks as data structures

A rule object can be created and optionally modified (disabled, reconfigured, etc.) at run-time. Rules have a name, an antecedent defining a *before*-pattern, and a consequent defining the substitute. The pattern and substitutes are expression trees. Exactly how the patterns work is not discussed, but we imagine they are somewhat similar to the pattern matching described in Section 3.8. Exploration is done by successively comparing the before-patterns of the available rules, and applying the transformations of the rules that either match the input, or if a match can be created by exploration.

This way of incrementally applying rules on demand and continuously considering where to "go next" is in contrast to Volcano's strategy, which involved two phases: a first phase that applied all transformations to generate all possible logical expressions for a query and its sub-tree, with a subsequent phase that made physical plans from these and compared them to each other. The exploration is governed by heuristics that avoid repeatedly exploring the same sub-space. Also, *guidance*-instances can be created, whose function is solely to limit the search space. Without any guidance, the size of the search space explored equal that of Volcano. It is important that such guidance rules do not rule out potentially optimal plans. With sophisticated rules, the current optimization *goal* — such as cost and required properties — can be considered. Rules inform the optimizer of how *useful* they are in the current context, which affects the order in which the space is explored. Hopefully, this leads the exploration to the more promising subspaces first, which in turn can result in pruning large portions of the search space.

2.5.2 Transformative: Optimization of DAG-Structured Query Evaluation Plans

In "Optimization of DAG-Structured Query Evaluation Plans" [Roy98], Prasan Roy presents a few transformation rules and a transformative optimizer for DAG-structured QEPs based on the Volcano-optimizer from Section 2.5.1. It uses two steps. First, the operators that may be shared are identified. The ones that should be used are then duplicated, with the duplicates reporting their cost as 0, as it is paid the first time it is used. Then, a normal tree-based optimization is done.

In addition to the drawbacks about Volcano, Neumann identifies some issues with the approach [Neu05]:

- Identifying shareable operators must be done before the search phase. This is due to how the search phase is done with respect to "free" operators. Without care, the planner would only choose the free (duplicate) operators, and neglect the initial cost.
- The notion of multiple consumers causing no additional cost is only valid if nested-loop-joins are not considered. And *without nested-loop-joins, dependent joins and theta-joins cannot be performed* — only equi-joins!

The second limitation is very discouraging. It severely limits the scope of the optimizer. Even though MARS does not currently have nested loop joins, future versions might. Their importance cause this to be a severe limitation.

We are unaware of any other *transformative* approach to optimizing DAG-structured QEPs.

2.5.3 Constructive: Starburst and Neumann/Moerkotte

As mentioned, constructive optimizers determine the goal of a query, and then reconstructs it piece by piece. The optimization consists of finding cheap plans that solve sub-goals, and progress towards the penultimate goal, while retaining good sub-plans and avoiding spending too much time on bad ones.

In practice this involves ripping the query apart and reconstructing it in various ways. The disadvantage over transformative optimizers is that it is not easily possible to simply return the current best plan when e.g. the time budget has been spent. This is because the optimizer may only have reconstructed sub-goals of the query when the available time has been spent. Since hard optimization dead lines have not been a requirement, we have not looked further into this issue.

Starburst

We have not studied Starburst extensively, but Neumann's model is inspired by it, so we mention some important points. Starburst is the predecessor of the optimizer in IBM's DB2. The algorithm optimizes each operation in the query independently, bottom up. *Low-level plan operators* (LOLEPOPs), which operate on 0 or more streams of tuples and produce 0 or more new streams, are combined into *strategy alternative rules* (STARs). STARs have requirements its input plans must meet — for example, certain relations must be present, or the tuples must be in a specific order. If current plans do not meet these requirements, additional "glue"-LOLEPOPs may be added — such as adding a sort-operator when a certain ordering is required [PHH92].

Neumann and Moerkotte: DAG-structured Query Graphs

In his PhD-thesis "Efficient Generation and Execution of DAG-Structured Query Graphs" [Neu05], Dr. Thomas Neumann elaborates advantages with DAG-structured query graphs, identifies problems and presents his solutions and design of an optimizer. The work is continued in "Single Phase Construction of Optimal DAG-structured QEPs" [NM08], in collaboration with Prof. Dr. Guido Moerkotte. ***It is this model we have based our optimizer on.*** We describe the model in more depth in Chapter 3, where design- and implementation details are discussed. There, we also point out some differences and claim a few improvements to the original model. In Chapter 8, we point out some limitations and issues we have not yet had time to solve.

In short, the optimizer assigns instances of applicable rules to every node in a logical query graph. A rule has a set of **properties** it *requires* from its inputs, and a set of properties that it *produces*. The query's *goal* is also expressed as a set of required properties, and a plan is semantically equivalent when the produced properties is equal to the goal. The constructive approach has some advantages to transformative approaches when dealing with DAGs — for example, finding sub-plans with output that are share equivalent with the required input is easier. Two expressions are **share equivalent** *if one expression can be computed by using the other expression and renaming the result* [NM08].

Top-Down or Bottom-Up?

There are two approaches to constructing queries — either beginning at the top, requesting sub-goals recursively; or at the bottom, starting with the base relations (such as table- and index scans) and progressing towards the ultimate goal. In [Neu05], Neumann mentions three advantages with the top-down approach:

- Rules are more intuitive when written in a top-down manner, as with a top-down parser. This eases development and maintainability of the rule set.
- The planner quickly learns solutions to sub-problems. This helps establishing cost boundaries early on in the process, which is an important factor in reducing the search space. [Neu05] mentions experimental results showing a 10-20% reduction of the search space size.
- By recursively requesting plans satisfying specific properties (the sub-goals), only sub-plans satisfying sub-goals of the top goal will be considered, as opposed to the bottom-up approach which tries any combination.

However, the top-down approach will consider lots of plans that are not actually *possible* to execute. A substantial amount of time is spent trying to solve sub-goals which have no solution — a problem the bottom-up approach does not have. Neumann claims that for chain-queries with $\geq$10 relations, $>$99.9% of the time is spent on unsolvable sub-problems, if this problem is not mitigated. It is easily remedied by checking if the sub-problem is actually solvable. However, the check is still $\mathcal{O}(n)$ for n operators. Neumann states that $\geq$90% of the CPU time is spent on this check. In Figure 8.7 on page 136, a profiling run of our optimizer prototype is shown. *QueryOptimizer.GoalIsUnreachable(goal)* is the name of the check in our system, in which approximately 7% of the time is spent. As we explain in Section 3.6, the problem is ameliorated by caching the answers to *GoalIsUnreachable(goal)*. By disabling the cache, a large query which took 0.93 seconds to optimize with caching took 2.15 seconds.

Another advantage with the bottom-up approach is avoiding a lot of cache lookups for the same sub-problems when constructing DAGs.

With the caching of the reachability-check, the differences between bottom-up and top-down boils down to the overhead of numerous unneeded hash table lookups. We decided time is better spent on other issues than optimizing the amount of lookups. *Therefore, we settled on the top-down approach due to clarity.*

2.6 Reflections

To summarize, the direction of the work shifted drastically when we discovered that the differences between tree- and DAG-structured query evaluation graphs were more profound than we initially believed.

After having surveyed research articles about query optimization and DAGs, as well as existing implementations, we realized we either had to start from the beginning, or to base our work on just a few works [Roy98, Neu05, NM08] that have yet to achieve much attention and peer review from the research community. Dr. Thomas Neumann, author and co-author of two of them, has this to say:

> "I found that it is nearly impossible to publish papers about optimizing DAGs. They are usually rejected with the argument 'already implemented in commercial database systems' :) This might explain the lack of research papers."

In a later email he points out that "already implemented in commercial database systems" refers to the use of temporary relations as discussed in Section 1.7.2.

In Section 2.5.2, we mentioned some severe limitations in [Roy98]'s approach that handles DAGs, and that basically rules out the only real alternative to Neumann and Moerkotte's work. All in all, our impression is that Neumann and Moerkotte's model in [NM08, Neu05] is a lot more solid than that of [Roy98]. This is not surprising, as [Roy98] is a master's thesis,

whereas [Neu05] is a PhD-thesis, with subsequent work in [NM08]. Furthermore, it would be unreasonable to believe that an attempt to modify and extend e.g. Cascades to support DAGs would have yielded better results than a PhD devoted to the topic.

Thus, we have chosen to largely base the optimizer model on Neumann and Moerkotte's work. Chapter 3 and the rest of the report is devoted to our changes to the model and the design- and implementation details.

3 Design and Implementation

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

— C.A.R. Hoare, The 1980 ACM Turing Award Lecture

3.1 Introduction and Goals

In this chapter, we present the design of our optimizer. We start out by giving a high level picture of its design, before we delve into inner workings and implementation details for each step.

If we were to make the readers completely understand the design with one single pass of this chapter, it would be a remarkable feat. More likely, re-reading this chapter after having read the complete walkthrough in Chapter 7 is a good idea. Furthermore, the cost component, orderings- and groupings-manager and rule specifics are covered in depth in separate chapters.

The goals of the optimizer *implementation* are as follows:

1. Extensibility
   (a) Support for arbitrary operators and their optimizations
   (b) Support for arbitrary cost models for operators
   (c) Support for arbitrary pre- and post-processing.
2. Support multi-queries and shareable sub-plans, i.e. DAG-structured query plans.
3. Clean design and implementation
4. Efficient plan generation

Extensibility is the most important goal of the thesis. The optimizer should provide support for arbitrary operators, meaning that it should be able to add optimization rules for operators added after the optimizer was originally designed — and not impose any restrictions on what optimizations are possible. To achieve this, the optimizer cannot know anything specific about the operators, but uses rules created by the operator implementer instead. Specifics about operator semantics should be confined to the rules implementing them, with as few inter-dependencies as possible, to have a clear separation of concern.

Furthermore, different operators can have very different cost models. The operator implementer should be able to specify the cost model (e.g. how costs increase with tuple size), and have the optimizer adhere to it. Custom rewriting steps should also be supported.

Support for DAG-structured query plans, and thus the ability to share sub-plans and execute several queries simultaneously, is a defining feature of MARS. Thus, creating an optimizer that handles this is a requirement for the optimizer to be useful for MARS — and to contribute with something new.

We believe it is more important to come up with a good architecture and a clean design than to implement as much functionality as possible. Therefore, we have not implemented support for all of MARS' operators, but just a few important ones — and focused on the optimizer core.

Efficiency has not been highly prioritized. Clearly, design decisions that prevent efficient execution must be avoided, but we have not delved deeply into optimizing the implementation. Not only do such optimizations often result in less maintainable and readable code, but the code to be optimized should also be subject to real workloads to identify the real bottle-necks — without which we would just have spent time on premature optimization. We have gone to some lengths to have efficient algorithms in the inner loops of the search phase, though, as that is clearly the most performance critical path in the optimizer.

3.1.1 Testing

Testing is an important part of software development and needs to be carried out to make sure what is being developed works as expected. This is certainly true for our optimizer as well. Important topics include:

Optimization results. Given the constraints of the query and search space (only left-deep plans, for example), the optimizer should produce the optimal plan.

Time spent optimizing. The optimizer should not spend significantly more time than expected to optimize a given query.

Dependency injection. An important enabling factor to thoroughly unit test, is to have clear dependency boundaries. Being able to easily mock and stub depended-on components ease testing.

To make sure that our optimizer works as expected, we have employed automated testing by using the *NUnit* test framework [NUn08] and have implemented several automated tests. The tests create a query to be optimized programmatically and then invoke the optimizer. Afterwards, they verify that no exceptions occurred and that the resulting query is the optimal one. The tests have also been used to generate the results found in Section 8. Full test coverage has not been a high priority, but *testability* has. We have 87 tests of varying quality, ranging from unit- to smoke tests.

As an example, we have included the code for one of the plan generator tests in Section B.6.

3.2 The Big Picture

As mentioned in Section 1.3, query optimization consists of several steps, of which each may consist of several phases. We have focused on the rewrite and planning steps of the process, and not on the parse, analyze and execution steps. Nor have we focused much on the "housekeeping" procedures involved in query optimization, such as plan caching and cache invalidation. Such components depend heavily on the run time system, and is outside our scope. As such, Figure 3.1 shows the parts of the query optimizer that actually optimize the query, where our focus has been.

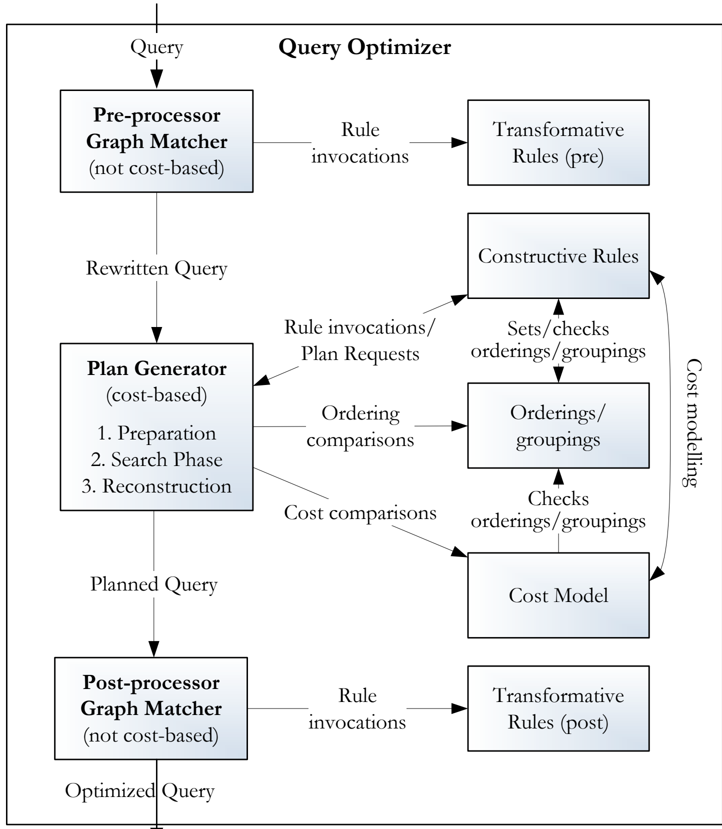


Figure 3.1: Query optimizer overview

The optimizer is invoked by a component that is injected into MARS's query pipeline, directly before the query is passed on to the evaluation operator. It handles conversion from and to MARS's graph structure.

3.2.1 Before Optimization

Before the optimizer is invoked, the query has gone through parsing and semantic validation. The input to our optimizer is a physical query operator graph. It is important to note that the query language currently available in MARS — MQL, Mars Query Language — is not a declarative language. It expresses query operator graphs. The availability of a query optimizer will likely ease development of a declarative language, as an optimizer can replace logical operators with suitable physical ones. We actually convert the physical join operators into logical ones during pre-processing to work around this limitation.

The input operator graph is converted to our own graph structure — which is easy to traverse, modify and apply pattern matching to. This graph structure is what the pre- and post-processing steps operate on — and what we convert MARS operator graphs to and from. The rationale for having our own graph types is not only because we did not have access to MARS's source code, but we also wanted to emphasize that the optimizer is not necessarily tied to MARS specifics.

3.2.2 Pre-/post-processing vs Plan Generation

The unoptimized query enters the optimizer in the upper edge of Figure 3.1, going straight into the pre-processing step. We have chosen to call the rewrite steps pre- and post-processing as they happen before and after the main step: Cost-based plan generation, also called the

search phase. *The distinction between pre-/post-processing and plan generation is very important.* While pre-/post-processing is transformative and linear and not cost modeled, plan generation is constructive and combinatorial and cost modeled. In other words, pre-/post-processing applies matching rules successively, generating a (mostly) linear chain of *equivalent* query graphs. It may generate several equivalent, rewritten plans during processing, but the number will be far less than what plan generation does. Plan generation *searches* for the cheapest plans by combining operators in many different ways, resulting in possibly many millions of smaller plans that are retained in memory, all sub-problems of the complete query.

Since the plan generation step is more computationally expensive and memory intensive than pre/post-processing, this should motivate us to try and do as much as possible in pre-/post-processing and only do what is strictly necessary in the plan generation step. Adding unnecessary search rules to the plan generation step will increase the size of the search space unnecessarily and increases memory usage and query optimization time. However, optimization strategies/rules that need to have costs modeled will usually have to be included in the plan generation step. For most queries, it is expected that *the bulk of the time will be spent in the plan generation step*.

3.2.3 Optimization Steps

In the **pre-processing** step, transformations like view flattening or predicate push-down or pull-up [LM94] are performed. Rewrites are done by applying transformation rules that are triggered when the transformation rule's pattern matches the input query graph.

After pre-processing, the operator graph is passed on to the **plan generation**[1] step. Plan generation is the step that we traditionally perceive as query optimization. It reorders operators, enumerating many plan alternatives, all the way evaluating and pruning them using the cost model. Internally, the plan generation step consists of three phases: (1) preparation, (2) search and (3) reconstruction. The **preparation phase** prepares for constructive plan generation. It analyzes the operator graph and instantiates applicable constructive rules and configures them. This includes looking up the possible useful access paths for the query and preparing the orderings and groupings component.

The logical operators in the query are also examined to determine the *logical goal* of the query, which is what our construction based optimizer uses as its starting point. If the query is a multi-query, this is done for each query in the multi-query. The goal is expressed as a *query goal property set*. Examples of properties in this set include "attribute X available" and "operator Y applied", but can be modeled to express anything. Each instantiated rule also have `Required` and `Produced` property sets which are also determined during this phase. Property sets are explained in Section 3.6.1. The `Required` property set describes operator dependencies in addition to what sort of input it requires, while `Produced` describes what the operator produces. These property sets in combination with their rules are all the optimizer uses. The query graph itself is disposed of and not used during the search phase.

The **search phase** is the heart of the optimizer and is where the actual cost-based planning is performed. A top-down, recursive strategy is used where the constructive rules control the direction of the search. Basically, solutions to sub-problems are combined into solutions of larger problems and finally the whole query, but in a top-down fashion. *Memoization* is used to avoid duplicate work and cost dominated plans are pruned along the way.

After the search phase has determined the best plan, the **reconstruction phase** translates the plan structure back into the node structure used by the post-processing step. This is carried

[1] The *plan generator* is the component carrying out the *plan generation* step, while the *query optimizer* is the complete optimizer.

out recursively by the rules themselves.

As Figure 3.1 illustrates, the **cost model** is an external component and not internal to the planner. Both the rules and planner have a well-defined interface to the cost-model, which allows for custom and extensible cost model implementations. More on this in Chapter 4. To keep track of different plans' useful orders and groupings an **ordering and grouping component** is used. In short the optimizer asks the ordering/grouping-component if an input plan satisfies a certain ordering/grouping, for instance when inserting a `MergeJoin` operator. If not, it can choose to add a sort operator. How this works is quite complicated, so Chapter 6 is dedicated to it.

Then, the planned query is **post-processed**. This is similar to pre-processing, but on physical algebra and typically, other types of rewrites are performed.

Lastly, the optimizer's graph structure is converted back to a MARS query graph, and the query continues in MARS' processing chain, most likely directly to evaluation. This is done outside the optimizer itself, by the MARS-component that invokes the optimizer.

3.3 Node Structure

The optimizer needs to work with query graphs in memory, and therefore needs a data structure to model such graphs. We considered directly operating on the MARS graph of operators implementing the IOperator interface found in MARS, but abandoned this since we wanted to be able to extend each node in the graph with custom properties needed in optimization. If we were to use the classes from MARS, we would need to extend each of the different operators, which would be neither extensible nor maintainable. Instead, we implemented our own Node and OperatorNode classes to model the graph, shown in Listing 3.1. Using a separate class also makes it easier to test the optimizer separately from MARS — or to later on use the optimizer in another product. The Node-class is the base for several different graph structures, such as graph patterns and transformation chains described later on.

When a query is received from MARS, it will be handed to the optimizer as an IOperator graph. The optimizer traverses this graph from the root and translates it into a graph of OperatorNodes. OperatorType is set, parent and children relationships retained and all readable properties are copied into the `Properties` dictionary. When the optimization is complete, the OperatorNode graph is be translated back to a MARS IOperator graph.

3.4 Operator Dependency- and Equivalence Mapping

When optimizing queries, we want as much freedom as possible with regard to how and where we can move operators around. However, when doing so, we must be careful not to alter the semantics of the query. In short, if we model dependencies too stringent, we lose out on optimization opportunities — if they are too loose, the size of the search space explodes and/or semantics can be altered.

Figure 3.2 shows two quite simple query graphs. The Trim-operator reduces the number of tuples to the specified amount. In SQL, the query could have been `SELECT … FROM a JOIN b ON (…) JOIN c ON (…) LIMIT 10`. To achieve the optimized query, the dependency mapping is quite involved. First of all, we cannot say that joins depend directly on their input — if we do, we cannot pull out A and join it after B and C, because the first join of the input would have depended on both A and B as inputs. Furthermore, the trim needs special attention. To be able to move the join of A above the trim, we must be certain that the join does not alter the amount of tuples — if it did, the semantics of the query would have been altered, as we could have yielded more than 10 results. To be able to guarantee that moving the A-join as shown in

```
1  public class Node {
2     public List<Node> Parents { get; private set; }
3     public List<Node> Children { get; private set; }
4     public Dictionary<string, object> Properties { get; set; }
5
6     public object this[string name] {
7        get { return Properties[name]; }
8        set { Properties[name] = value; }
9     }
10
11    public IEnumerable<Node> GetTopologicalOrdering() { ... }
12    public virtual Node CloneNodeAndSubgraph() { ... }
13
14    // And so on. There are many utility methods for various graph operations.
15 }
16
17 public class OperatorNode : Node {
18    public Type OperatorType { get; set; }
19    public List<IRule> Rules { get; set; }
20    public Dictionary<string, NodeAttribute> AttributeOrigin { get; set; }
21    public HashSet<NodeAttribute> Dependencies { get; set; }
22    public IRecordSetTypeDescriptor OutputTypeDescriptor { get; set; }
23
24    public NodeAttribute OriginForAttribute(NodeAttribute attr) { ... }
25
26    // And various other properties and methods.
27 }
```

Listing 3.1: Node and OperatorNode classes (very simplified)

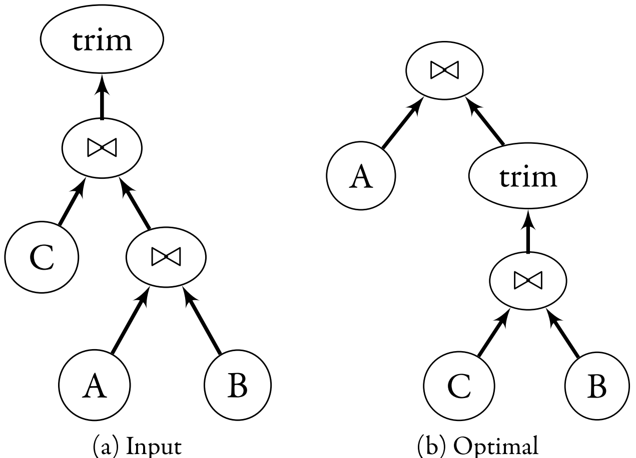


Figure 3.2: Simple Query Graphs with Advanced Dependencies

Figure 3.2b does not alter the amount of result, there must be a foreign key *to* A on the join key — so we are guaranteed that there is a 1:1-mapping between the relations. This information about relations is not readily available in MARS today, however, so we are currently unable to implement this optimization.

Dependencies are also used to constrain movement of outer join operators. Unlike inner joins, outer joins are not freely reorderable, so we need to "lock" them to be above or below the conflicting join. We cover this in Section 8.5.4.

Section 5.2.1 explains how these mappings are performed in their respective pre-processors. However, it is important to note that the mappings are entirely specific to the operators — the optimizer core's only responsibility is to invoke the mapping in the correct order.

3.5 Pre- and Post-Processing

Pre- and post-processing are mainly about query rewriting, but may also perform other tasks, like tagging the operator graph with information to be used later. In the **pre-processing** step, transformations like view flattening, sub-query merging/flattening, predicate push-down or pull-up [LM94] or different join transformations (like ([NOT] ANY|EXISTS) → (SEMI|ANTI)JOIN) are carried out. Other transformations that are "always smart to perform" are also carried out. Transformations removing unnecessary nodes can also be useful, as it reduces the running time of plan generation. Pre-processing operations that tag the nodes with auxiliary information for other processors and/or rules are also used.

Pre-processing does not necessarily result in a completely linear chain of transformations, as rewrite rules can generate several semantically equivalent operator graphs which differ greatly in optimal cost after plan generation. For example, the query `SELECT max(age) FROM foo` and `SELECT age FROM foo ORDER BY age DESC LIMIT 1` are equivalent — but the pre-processor cannot tell which one is cheapest. As [Neu05] explains, supporting such rewrite alternatives poses a challenge. Each alternative could be provided to the plan generator for full plan generation, but this is inefficient. They will probably overlap fairly much, which means that a lot of double work will be performed. [Neu05] suggests that the best approach is probably to have the rewrite steps during pre-processing not generate completely new operator expressions, but annotate parts of the existing expressions with alternatives. We have not had time to study these possibilities in depth.

Post-processing works on physical algebra. It is the last chance to alter the query graph before it is passed on for execution. Examples include merging of successive selections, maps, trim and sort and so on. Optimizations that do not need cost modeling and can be done independently from plan generation, should be considered for inclusion in pre- or post-processing to keep the search space size in plan generation down.

The processing is driven by a collection of rules that declares a `Pattern` it is looking for in the operator graph. When a pattern match is found, the processor's *Fire()*-method is invoked with the graph match(es), close to what was done in Starburst [PHH92]. *Fire* returns a boolean that indicates whether changes have been made to the graph. This way, non-transformation rules can be implemented in the same way as transformation rules — they just add information to the graph instead of altering it in the *Fire* method.

The interface for transformation rules is shown in Listing 3.2. For an explanation of how to express patterns, and what `GraphSearcher` is, see Section 3.8 on graph matching.

We recognize the need of some way of controlling the order of the applied transformations if multiple matches are found. At the same time, we want as few dependencies between the rules as possible. Therefore, each rule is allowed to expose a `DependsOn` property, listing the rules that should be run before this rule. At start up, all rules are topologically sorted by the

```
1  interface ITransformationRule {
2    AbstractNodeMatcher Pattern { get; }
3    bool Fire(GraphSearcher graphSearcher);
4    IEnumerable<Type> DependsOn { get; }
5    bool Iterative { get; }
6  }
```

Listing 3.2: ITransformationRule interface

```
1  public void Transform(Node root) {
2    // ... declarations ...
3    do {
4      graphHasChanged = false;
5      foreach (ITransformationRule transformer in TransformationChain) {
6        if (completedTransformationRules.Contains(transformer))
7          continue;
8        GraphSearcher matcher = transformer.Pattern.GetSearcher();
9        if (matcher.Search(root)) {
10         graphHasChanged |= transformer.Fire(matcher);
11         if (! transformer.Iterative)
12           completedTransformationRules.Add(transformer);
13       }
14     }
15   } while (graphHasChanged);
16 }
```

Listing 3.3: *Transform* method

optimizer and serves as a foundation for rule invocations. We have not had need for it yet, but it is also easy to implement having a `Precedes`-attribute that declares which transformers should be run *after*. Any cycles in the dependency graph is clearly a development error — in which case an exception is thrown.

Each rule may be applied more than once. For example, a rule merging two adjacent nodes may be run twice to merge three adjacent nodes. The optimizer successively applies rules until the graph converges to a final stable result. If the rule returns *true* for `Iterative`, the optimizer will continue invoking the rule until the graph stabilizes. This approach should be avoided, but is preferred if it greatly simplifies the rule's pattern and/or transformations. Care must also be taken to avoid a "ping-pong" situation where the graph is transformed back and forth. Listing 3.3 shows how the transformers are successively invoked.

Implementations of a few pre- and post-processors are covered in Section 5.2

3.6 Plan Generation

The design of the plan generation step is based on the work in [Neu05] and [NM08]. Many of the design principles described in this section are close to what is described in these two works. Instead of repeatedly citing these two works, we point out when our design differs significantly from theirs. However, we claim to have produced a few improvements to the design. These are detailed in Section 3.6.7.

In the plan generation phase, each rule constructs one part of the query. Each rule usually represents one logical operator, but it does not have to — it can be used to construct more than one operator (e.g. Join, which can create the different join operator implementations). Each rule appliance creates a *plan* that is a solution to a sub-problem of the whole query, and

```
public OperatorNode Optimize(OperatorNode query) {
    // Initialize managers.
    BitSetManager = new BitSetManager();
    OrderManager = new OrderManager()
    // Preprocess the graph.
    Preprocess(query);
    // Find and instantiate rules.
    InitializeRules(query);
    // Prepare bit sets.
    DeterminePropertySetsAndPrepare();
    // Determine share equivalence classes.
    constructShareEquivalenceClasses();
    // Find out the goals of the sub-plans.
    DetermineGoals(query);
    // All requested and produced orders have now been set. Prepare orders.
    OrderManager.ShareEquivalentMappings = getShareEquivalentOrderings();
    OrderManager.PrepareOrders();
    // Instantiate the memoization table, using a rough estimate for entries.
    plansCache = new Dictionary<BitSet, PlanSet>(BitSetManager.Count * 10000);
    // Make plans for leaf-nodes, i.e. scans.
    InitializeBasePlans();
    // Go plan!
    GeneratePlans(query);
    // Convert plans back to nodes.
    query = MakePhysicalPlan(query);
    // Do any post processing.
    Postprocess(query);

    return query;
}
```

Listing 3.4: Query optimizer main method, simplified

finally the complete query. Each plan can have a number of *sub-plans*, solving a sub-problem of the plan's problem — i.e. its input.

Listing 3.4 shows the main method of the query optimizer, somewhat simplified. It closely resembles what was described in Section 3.2. First, a new BitSetManager and OrderManager is created to handle the property sets and orderings for this query. Then the query is pre-processed, rules relevant to this query are initialized, property sets prepared, sharing (DAG) opportunities found, goals determined, orderings and groupings prepared and base plans added, all as part of the preparation phase. Next, the search phase starts with the call to *GeneratePlans*, before the final plan is reconstructed, post-processed and returned.

To be able to illustrate how the plan generation works, we now introduce a simple query. It is a simple join of two relations with a selection on an attribute of the second relation, expressed in SQL below. Figure 3.3 shows how this query might be passed to optimizer for optimization. The topmost operator is MARS' output operator.

```
SELECT * FROM A
    JOIN B ON A.a1 = B.b1
    WHERE A.a2 = 8
```

In this simple query, the only optimization possible is probably to push the selection through the join, provided that the predicate is not very expensive or the join very selective. Still, it serves as a good example.

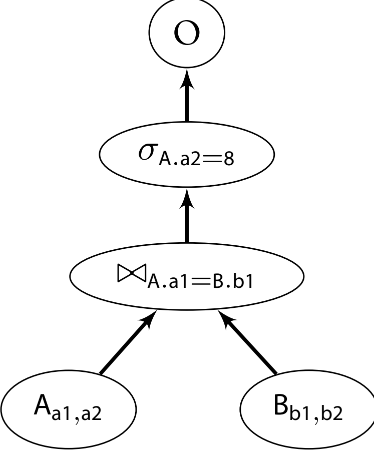


Figure 3.3: Sample query

Id	Type	Requires	Produces
1	Lookup:A		a1,a2
2	Lookup:B		b1,b2
3	Join	L:a1 R:b1	$\bowtie$
4	Selection	a2	σ

Table 3.1: Rules and produced-/required sets

3.6.1 Property Sets

The plan generator produces many plans as solutions to sub-problems. In some way, it needs a way to annotate these plans with what they actually output — attributes available, ordering, operators applied, relations involved and so on. This could be solved as a list of operators applied and attributes available, but this is not very extensible. Instead, we use the model proposed in [Neu05], where it is modeled as a set of general properties. Examples of properties are "attribute X available" and "operator Y applied" — they can be used to express anything. The plan generator does not know anything about their meaning, it only cares about satisfying them during plan generation. It is up to the rule implementer to define their meanings.

Properties are also key to how the operator uses rules to construct plans. As our optimizer is constructive, different combinations of rule instances are used to produce plans with different properties. Each rule declares several property sets: A `Produced` set and a `Required` set for each input. A rule usually produces that itself was applied and requires the attributes it operates on. The required and produced sets for the example query in Figure 3.3 are given in Table 3.1. For example, the join requires {a1} for its left input and {b1} for its right, since the join predicate includes these two attributes. It produces {$\bowtie$}, i.e. that itself was applied.

The *global query goal* is the property set the final, complete query plan should satisfy. It serves as the starting point for plan generation. It is computed as the union of the produced sets of all the *logical* operators in the query. In the example query the global query goal is {a1,a2,b1,b2,$\bowtie$,σ}.

Implementation

[Neu05] does not explain how to identify the different properties in a concrete implementation. We propose to identify them by string constants, as this allows for arbitrary expressiveness. For example, properties for attributes and operator applied would translate to "`AT_X_Y`" (attribute Y from operator X) and "`AP_X`" (operator X applied).

As such, each property set will be a set of strings. Each plan, as well as all rules would include such sets. As the plan generator will produce potentially millions of plans, it is desirable that the property set is as *compact* as possible to consume little memory. A set of strings is certainly not compact. Also, as we will see, the search phase performs a lot of set operations like union and intersection on these sets. Therefore, we should optimize for this.

A key observation is that the universe of possible properties is determined *before* the search phase begins. As [Neu05], we therefore assign a bit to each possible property and simply store

```
1  public struct BitSet : ICloneable {
2     void Add(string property);
3     bool Contains(string property);
4     void Remove(string property);
5     bool Overlaps(BitSet other);
6     static bool operator ==(BitSet a, BitSet b);
7     static bool operator <=(BitSet a, BitSet b);
8     static BitSet operator |(BitSet a, BitSet b);
9     static BitSet operator &(BitSet a, BitSet b);
10    static BitSet operator -(BitSet a, BitSet b);
11    void Or(BitSet other);
12    IEnumerable<BitSet> Walk();
13    int GetHashCode();
14    bool Equals(Object other);
15 }
```

Listing 3.5: BitSet struct (simplified, interface only)

the property sets as bit masks. This allows us to store 8 properties in just 1 byte. See Section B.5.1 for details. Further, this allows for *very* fast set operations, as set intersection is simply a bit wise AND between two property sets, performed in just a few CPU instructions. See Listings B.8 and B.9 in Section B.5.1 for code snippets showing how the intersection- and subset operators are implemented.

The BitSet struct is the implementation of a property set (*Bit*Set because of the storage mechanism). Structs in C# are stored directly on the stack, not on the heap with a pointer to it, allowing for faster access. Listing 3.5 shows the interface to the BitSet struct. We find most of the usual operations for sets, including operators for set equality/inequality, union, intersection and subtraction. The *Or* method allows for in-place union.

Worth noting is the *Walk* method that returns a sequence of BitSets. It returns all permutations of the properties in the current BitSet and is used by the join rule, as we will see later.

To conserve space, each BitSet does not store the mapping *string* → *bit index*. That is the responsibility of the BitSetManager class. How the *Add-*, *Contains-* and *Remove* methods use it is shown in Listing B.6 in Section B.5.1.

BitSets are used as keys in the memoization table, and as such they also implement *GetHashCode* and *Equals*.

BitSet Minimization

Another responsibility of the BitSetManager is *BitSet Minimization*. Often, declared bit properties turns out to never be produced or never required and are therefore removed. Furthermore, if some properties are always produced together, they are merged into a single property and represented with a single bit, saving memory and reducing the search space. We have only implemented the latter, and the code can be found in Section B.5.2.

Proxy Technique

It may not be clear how an instance of a BitSet is acquired, since the BitSetManager will need to know all the BitSets produced and required before it can minimize them, which again need to happen before any BitSet instances can be produced. Therefore, when acquiring a BitSet by specifying a set of string properties, one does not get a BitSet, but a BitSetProxy as shown below (simplified).

```
BitSetProxy BitSetManager::GetBitSet(HashSet<string> properties)
```

A BitSetProxy is a simple class which serves just one purpose; acting as a proxy for the BitSet requested. More specifically, when requesting a BitSet, the BitSetManager will return a BitSetProxy with `Properties` set to the requested properties, but with the BitSet property empty. It will also *keep references* to all returned proxies. Later, when all rules have requested their BitSets and the manager has minimized them, it will iterate over all the proxies and update the `BitSet` property with the appropriate BitSet.

This is much more elegant than having a two-phase approach to BitSet acquisition and does not affect performance adversely since the number of proxies is quite low — it is the number of *calculated, intermediate* BitSets during plan generation that is high.

3.6.2 Other Properties

When dealing with sub-plans, there are other properties that need to be considered than their *produced* bit set or raw cost. Costs are just half the story — properties such as ordering, groupings, and sharing are examples of others. Keeping plans that allow more sharing even though they are more expensive, is key to finding plans that are *globally* optimal.

When taking this into consideration, there is seldom such a thing as a "best" sub-plan. One sub-plan may be more expensive than another one, but may offer a more useful ordering. Therefore, the plan generator often needs to retain multiple plans satisfying the same properties. Although orderings, groupings and the like can be expressed as bit properties, we need more reasoning capabilities than they offer, and the representation we will present is also more compact. Costs are discussed in Chapter 4, sharing in Section 3.7, while orderings and groupings are covered in depth in Chapter 6.

When dealing with ranked queries, *ranking* properties may also be interesting to track, but we have not studied this in depth.

3.6.3 Data Structures

Memoization Table

To avoid solving the same sub-problem more than once, the plan generator memoizes solutions to problems solved using a hash table. A set of properties uniquely identify a problem as previously discussed, so BitSet is used as key in the table. Since multiple plans can satisfy the same properties (they can have different orderings or not dominate each other in other ways), each table entry stores a PlanSet, not a Plan. This table is the primary memory consumer in the optimizer, but our testing has discovered that it is in the area of up to 50 MiB for moderately sized queries (7–8 relations).

The memoization table is declared in C# as a `Dictionary<BitSet, PlanSet>`.

PlanSet

Multiple plans satisfying the same properties are organized in a PlanSet, which contains data common to the plans. As such, a PlanSet can be viewed as a container for the solutions to a sub-graph of the entire solution operator graph (the equivalent for trees would be a *branch*). For example, a PlanSet containing plans (or actually the single plan in this case) that have only applied the selection in the example query earlier in this section would have the properties $\{b1,b2,\sigma\}$.

PlanSet prunes dominated plans whenever they are added. Thereby, the set will never contain any plan dominated by any other plan (e.g. in terms of cost, sharing and ordering), and hence the optimizer will never store such plans.

```
1  public class PlanSet : IEnumerable<Plan> {
2     private List<Plan> plans = new List<Plan>();
3     public BitSet Properties { get; set; }
4     public IPlanSetState State { get; set; }
5
6     public void AddPlan(Plan planToAdd) {
7        // Simplified: Add plan to PlanSet if not dominated by existing
8        // plans in the set, removing any plans it dominates.
9     }
10    internal Plan GetCheapest() { // Simplified: Return cheapest plan. }
11 }
```

Listing 3.6: PlanSet class (simplified)

```
1  public class Plan {
2     public Plan(params Plan[] children);
3
4     public PlanSet PlanSet { get; set; }
5     public IRule Rule { get; set; }
6     public List<Plan> Children { get; set; }
7     public OrderingState OrderingState { get; set; }
8     public ICost Costs { get; set; }
9     public BitSet Sharing { get; set; }
10    public bool Shared { get; set; }
11
12    public PlanRelation Compare(Plan other);
13    public PlanRelation CompareTotal(Plan other);
14 }
```

Listing 3.7: Plan class (simplified, public interface only)

The data structures and methods are shown in Listing 3.6. The list `plans` contains all the plans currently stored. `Properties` stores the set of properties satisfied by all the plans in the set. `State` stores the logical state common to the plans, as defined by the cost model (e.g. cardinality and tuple size). Finally, the *GetCheapest* method is used by the optimizer to get the cheapest plan in the set when constructing the final plan in the reconstruction phase.

Plan

The Plan class represents a concrete plan that satisfies the properties of the PlanSet containing it. Concrete in the sense that it has physical operators, an actual order of the operators and thereby an estimated cost. It actually represents the topmost node in the plan, containing references to its sub-plans. It is as compact as possible, as potentially many millions will be created and references the creating rule for the details needed when reconstructing it to an operator graph.

The `PlanSet` property contains a reference to the enclosing PlanSet, while `Rule` contains the rule that constructed this plan. The list `Children` contains all sub-plans of this plan, listed from left to right. For example, for a two-way join, this list contains the left and right input to the join. `OrderingState` contains the logical ordering and grouping state of the plan and is explained in Section 6.2.5. Rules/operators affecting the order (like the example above) will update this property to reflect it, while rules/operators requiring some order can consult it to find out if it is satisfied. If not, they can explicitly insert a sort operator. `Costs` stores this plan's costs, as explained in Section 4.4.1. `Sharing` stores sharing opportunities (for DAGs), while `Shared` stores if this exact plan is shareable. This is covered in Section 3.7. The *Compare*

```
public interface IPlanSetState { }
public class BasicPlanSetState : IPlanSetState {
   public double Cardinality { get; set; }
   public double TupleSize { get; set; }
   public double ResultSetSize { get{ return Cardinality * TupleSize; }}
}
```

Listing 3.8: BasicPlanSetState class (simplified)

and *CompareTotal* methods enable the optimizer to compare plans based on cost, sharing and ordering.

PlanSetState

To be able to calculate costs, the cost model needs some state information common to all plans satisfying a set of properties. Therefore the PlanSet stores an instance of IPlanSetState. It is up to the cost model to define what it wants to store, so the definition of IPlanSetState is empty. The optimizer core does not care about it contents.

For our simple cost model, we model the plan set state with the BasicPlanSetState class. For each set of plans satisfying the same set of properties, we keep track of the expected *cardinality* and *tuple size*. It is logical that all plans producing the same result will have the same cardinality and tuple size; otherwise the techniques used for estimating these sizes would be broken.

3.6.4 Preparation

The preparation phase of the plan generation step has several responsibilities and this list closely resembles the code in Listing 3.4.

Instantiate applicable constructive rules. The input query operator graph is analyzed, and any rules found to be relevant will be instantiated.

Configure rules. The rules instantiated in the previous step are configured. The `Produced` and `Required` properties are set, and any other properties specific to the rules are set. For example, the selectivities of joins and selection predicates are looked up and set using statistics. For the example query introduced in Figure 3.3, Table 3.1 shows the instantiated rules and their `Produced` and `Required` properties.

Minimize bit properties. As described in Section 3.6.1, bit properties are minimized during this phase.

Construct share equivalence classes. The constructive rules are compared with respect to share equivalence and all share equivalent rules are gathered in equivalence classes.

Determine goals. For each query (the query can be a multi-query), the logical operators in the query are examined to determine the *logical goal* of the query. This is what our construction based optimizer uses as its starting point.

Prepare orderings. At this step, all rules have determined their produced and required orderings, so the order manager is told to generate the finite state machine to be used during plan generation.

Initialize memoization table. The memoization table is initialized with an estimated number of entries.

Look up access paths. The possibly useful access paths for the query is looked up using the system catalog. For example, if a query is found to access relation A, table- and index scans rules are added for A and their expected tuple sizes and counts are looked up. If the query includes a selection, it can also choose to combine an index scan and the selection in one rule.

Add base plans. *Base plans* are plans produced by IBaseRule rules as described in Section 5.3.1. Base rules are rules representing table and index scans. They produce properties, but have no requirements. As such, base plans are usually the leaves of the operator graph. They are computed and entered into the memoization table before the search phase begins, as they do not provide search facilities, they only serve as a foundation to the plans generated.

3.6.5 Search

Introduction

The search phase of plan generation is the heart of the optimizer, and is where the actual cost-based planning is performed. A top-down, recursive strategy is used where the input to each recursive call is a property set to be satisfied, the *local goal*. The signature of the main method of the search phase, *GeneratePlans*, is as follows:

```
public PlanSet GeneratePlans(BitSet goal, ICost limit);
```

When called, it will generate the best plans that satisfies the BitSet specified for `goal`, the local goal. More than one plan can be returned (e.g. due to different orders). `limit` is used for cost-based pruning, aborting the search as soon as the cost reaches `limit`. As such, time is not spent on constructing plans that is guaranteed to be dominated by another plan. It returns a PlanSet with the best plans found for the given goal and cost limit. Any dominated plans have been pruned before the call returns.

Initially, the search is started with a call to *GeneratePlans* with the *query goal*, as determined during the preparation phase, supplied as the *local goal*. If the query is a multi-query, `GeneratePlans` is called once for each query in the multi-query. *Infinity* is supplied for *limit*, as we have no limit yet. This could be improved with heuristics — a better initial limit, such as the cost of the input query.

Currently, we do not actively use the `limit` pruning as described above, since it was a bit more complex than we anticipated. It is not as simple as using the cost of the cheapest plan found so far as a bound, since a plan that is more expensive could offer better ordering or more sharing. For orderings it is as simple as setting the limit to the cost of the cheapest plan + the cost of a sort (since a sort can satisfy all orderings and groupings). For sharing it is not so clear, since it is hard to know beforehand how many times a plan will be reused.

The plan generator does not call itself recursively, but leaves this to the search rules. *GeneratePlans* determines which rules are applicable and tries to use each of them to produce the requested goal. The rules themselves control the direction of search and call back to the plan generator, requesting solutions to sub-problems (subset of properties) as their input. Basically, *GeneratePlans* invokes the search rules, which in turn call *GeneratePlans* for their inputs. A very simplified version of our simplest search rule, the rule for creating selections, is shown in Listing 3.9 (the implementation is actually in UnaryRule, which SelectionRule inherits from).

If *GeneratePlans* finds that the SelectionRule may be used to produce a requested goal, it will call the *Search* method, asking the rule to produce plans with this goal, as set in the properties of *plans*, passed to the rule. The selection rule solves this by again requesting *GeneratePlans* to produce plans with the requested properties **minus** what the selection itself produces. Then

```
1 public override void Search(PlanSet plans, ICost limit) {
2    foreach (Plan input in qo.GeneratePlans(plans.Properties - Produced, limit)) {
3       Plan selectionPlan = new Plan(input) { Rule = this };
4       plans.AddPlan(selectionPlan);
5    }
6 }
```

Listing 3.9: *SelectionRule.Search()*, very simplified (actually *UnaryRule.Search()*)

it iterates over the received plans, adding itself as the top node of each of them. All of these plans are added to the supplied PlanSet, thereby returning them to plan generator. Note that any dominated plans are automatically pruned inside the *AddPlan* method, so no plan found to be dominated in terms of cost, ordering and sharing will be stored and used later in the search, effectively decreasing the size of the search space.

This effectively tries to add the selection to the top of any produced plan where it can be (its `Required` properties must be fulfilled). For example, a selection can not be put in a location where its filter attributes are not available. This is handled by *GeneratePlans* — a rule will not be invoked if it cannot be used.

To speed up the search, the optimizer *memoizes* solutions to sub-problems (actually partial query plans) with the property set produced by the plan as the memoization key. Thus, if the same sub-problem (the same property set) is requested from *GeneratePlans* twice, it will only be computed once. This reduces the complexity from approximately factorial ($n!$) to exponential (a^n).

Implementation

Listing 3.10 shows the implementation of *GeneratePlans*. First, on lines 4-5, the memoization table is consulted. If an entry is found for the specified goal, it means we have solved this problem before, and we just return the PlanSet stored in the entry. Next, a sanity-check is performed on lines 8-9 to determine if we can actually reach the requested goal with the rules available for use. This is explained below. If we cannot reach the goal, we store this in the memoization table as null and return null, as no plans can be generated. Memoizing this result is one of our proposed improvements to the implementation in [Neu05].

If we have reached this far, we know that we can construct plans, so we create a new PlanSet with properties set to the current goal on line 11. Then comes the part where DAGs are being constructed as described in Section 3.7. On line 14, it is checked whether the goal being sought can be completely rewritten to another, share equivalent goal. If so, a search for this rewritten goal is invoked on line 15. We use *infinity* as the cost limit as the shared plan is allowed to be more expensive, since it allows for more sharing. This limit can probably be improved for better pruning, though. Finally, if this search returns any plans, they are added to the current plan set on line 18.

Then, in the loop on lines 22-24, we try to apply all search rules, but only if it is relevant to the goal (have its requirements satisfied and produces a relevant property for the sought goal). We call *ISearchRule.Search*, supplying the plan set to which plans are to be added, and the cost limit. The rule will find the properties being sought in the plan set. Finally, we store the generated plans in the memoization table and return them.

The full, unsimplified version of *GeneratePlans* can be found in Section B.4.

```
1  public PlanSet GeneratePlans(BitSet goal, ICost limit) {
2     PlanSet plans;
3     // If we already have a plan that satisfies the goal, return it.
4     if (plansCache.TryGetValue(goal, out plans))
5        return plans;
6
7     // Check if we can reach this goal with the current rule set.
8     if (GoalIsUnreachable(goal))
9        return plansCache[goal] = null;
10
11    plans = new PlanSet() { Properties = goal };
12    // Rewrite using share equivalent representatives.
13    BitSet shared;
14    if (ShareEquivalentGoal(goal, out shared)) {
15       PlanSet sharingPlans = GeneratePlans(shared, BasicCost.Max);
16       if (sharingPlans != null)
17          foreach (Plan plan in sharingPlans)
18             plans.AddPlan(plan);
19    }
20
21    // Invoke search rules.
22    foreach (ISearchRule searchRule in searchRules)
23       if(searchRule.IsRelevantTo(goal))
24          searchRule.Search(plans, limit);
25    // TODO: Try to *lower* the limit-parameter for GeneratePlans, the problem being sharing.
26    return plansCache[goal] = plans;
27 }
```

Listing 3.10: *QueryOptimizer.GeneratePlans()*, simplified

```
1  private bool GoalIsUnreachable(BitSet goal) {
2     BitSet mask = BitSetManager.Empty;
3     foreach (IProducerRule producerRule in rules)
4        if (producerRule.Filter <= goal)
5           mask |= producerRule.Filter;
6     return mask != goal;
7  }
```

Listing 3.11: *GoalIsUnreachable()*

Rule Filters

A *Rule filter* is a relevancy check to verify if a rule is relevant to the goal being constructed. Each IProducerRule offers a property `Filter` that is usually the union of its `Produced` and `Required` properties. The *filter must be a subset of the goal being constructed to be relevant.* This is why: If its `Produced` property set is not a subset, the rule is useless. If its *Required* property sets are not subsets, the rule does not apply (requirements are not satisfied).

Therefore, before we start constructing a plan, we check if any combination of rules can construct the goal properties. If we did not do these checks, the time complexity of the search phase would be much greater, since we would be doing lots of unnecessary work. The algorithm can be seen in Listing 3.11.

Planner Interface for Orderings

The above discussed implementation of `GeneratePlans` does not produce plans with any specific ordering or grouping. Some rules, like `MergeJoin`, may want to request plans with a cer-

```
1 public PlanSet GeneratePlans(BitSet goal, ICost limit, Order order, IRule enforcer) {
2    PlanSet plans = GeneratePlans(goal, limit);
3    if (plans == null)
4       return null;
5    Plan planWithSort = new Plan(plans.GetCheapest());
6    enforcer.UpdatePlan(planWithSort);
7    plans.AddPlan(planWithSort);
8    return plans;
9 }
```

Listing 3.12: *QueryOptimizer.GeneratePlans()* with ordering, simplified

tain ordering. Also, the `Output` operator will request a plan with a certain ordering if the query specifies an output ordering. Therefore, the plan generator also includes an overload to `GeneratePlans` which will produce plans with a requested ordering.

Listing 3.12 shows its implementation. In addition to the usual arguments, it also takes an order that the plan should satisfy and an enforcer rule (which in most cases will be a `Sort` rule) the plan generator can apply to produce the requested ordering. It works by calling the usual `GeneratePlans` method, but will afterwards fetch the cheapest plan from the results and add the enforcer rule get at least one plan with the requested ordering. Note that currently, the `order` parameter is not used since `PlanSet` will handle the pruning automatically. However, it is still there for interface completeness.

Sample Search

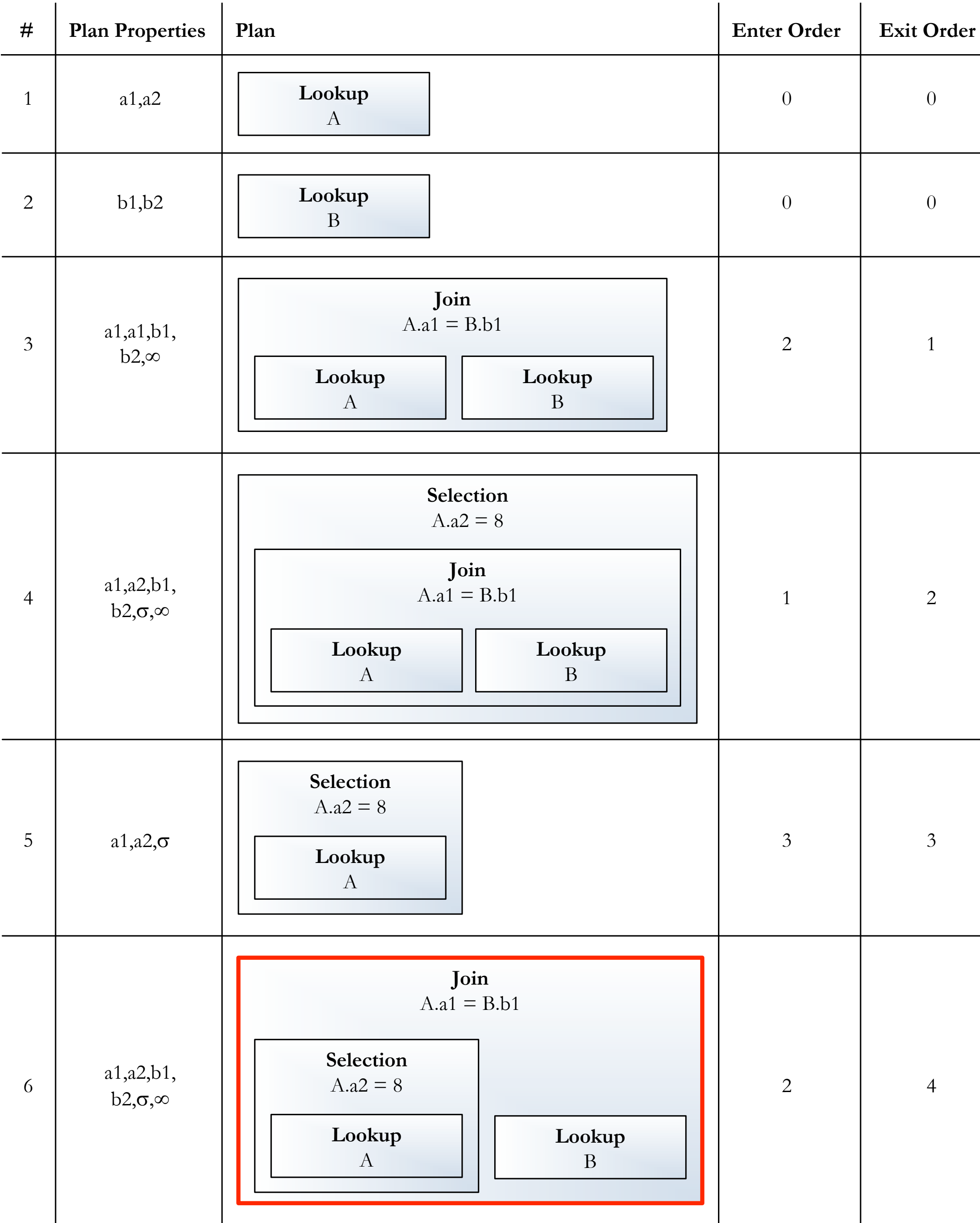
The top two rules in Table 3.1 are the instantiated base rules for our sample query, while the bottom two are the instantiated search rules. Assuming the selection rule is applied first, the plans shown in Figure 3.4 are generated during the plan generation phase.

The *Plan Properties* column lists the properties produced by the plan, while the *Enter Order* and *Exit Order* shows the order in which the plan generator starts constructing the plan and when it finishes. Plans within other plans in the *Plan* column means that they are sub-plans. The plans are ordered by time of completion.

The two top plans are base plans initialized during the preparation phase. First the plan generator uses the selection rule to produce the goal, {a1,a2,b1,b2,σ,$\bowtie$}, by starting the construction of plan 4. This again triggers the construction of plan 3 by using the join rule. The join rule will find its inputs (the base plan) in the memoization table as they were added during preparation. When finished, plan 3 becomes a sub-plan of plan 4. Then, it tries to construct the goal by using the join rule instead of the selection rule, thereby starting the construction of plan 6, which triggers the construction of plan 5, which when finished becomes a sub-plan of plan 6. Both plans 4 and 6 are complete plans, but in this case, plan 6 is chosen because of lower cost (not shown). Note that the selection put itself on top of two plans — the base plan and the join plan, effectively producing all possible plans.

Binary Rules

To give a short taste of how a binary operator might be implemented, we have included a very simplified version of our join rule in Listing 3.13. The full implementation can be found in Section 5.3.9. The rule works by determining all the properties that either of its children must satisfy. This is `wantedProperties` = `Requested properties` - (`Produced` | `RequiredLeft` | `RequiredRight`). Then these properties are distributed between the left and right sub-plan in all possible ways, plans requested from the plan generator and added to the plan set.



#	Plan Properties	Plan	Enter Order	Exit Order
1	a1,a2	Lookup A	0	0
2	b1,b2	Lookup B	0	0
3	a1,a1,b1, b2,∞	Join A.a1 = B.b1 (Lookup A, Lookup B)	2	1
4	a1,a2,b1, b2,σ,∞	Selection A.a2 = 8 (Join A.a1 = B.b1 (Lookup A, Lookup B))	1	2
5	a1,a2,σ	Selection A.a2 = 8 (Lookup A)	3	3
6	a1,a2,b1, b2,σ,∞	Join A.a1 = B.b1 (Selection A.a2 = 8 (Lookup A), Lookup B)	2	4

Figure 3.4: Plans generated for the query in Figure 3.3. The final plan is highlighted

```
1  public override void Search(PlanSet plans, ICost limit) {
2     BitSet wantedProperties = plans.Properties - (Produced | RequiredLeft | RequiredRight);
3
4     foreach (BitSet left in wantedProperties.Walk()) {
5        BitSet right = wantedProperties - left;
6        foreach (Plan leftPlan in qo.GeneratePlans(RequiredLeft | left, limit)
7           foreach (Plan rightPlan in qo.GeneratePlans(RequiredRight | right, limit)
8              plans.Add(new Plan(leftPlan, rightPlan));
9     }
10 }
```

Listing 3.13: *JoinRule.Search()*, very simplified

```
public override Node BuildAlgebra(Plan plan) {
   Node newNode;
   // Check if we have already constructed this plan
   if (queryOptimizer.ReconstructionTable.TryGetValue(plan, out newNode))
      return newNode;

   // Call recursively
   Node input = plan.OnlyChild.Rule.BuildAlgebra(plan.Children[0]);

   // Create node
   newNode = new Node() { OperatorType = plan.Rule.Node.OperatorType };
   newNode.Children.Add(input);

   // Store in reconstruction table
   queryOptimizer.ReconstructionTable[plan] = newNode;
   return newNode;
}
```

Listing 3.14: *SelectionRule.BuildAlgebra()*, simplified

3.6.6 Reconstruction

After the search phase completes, we are left with a hierarchy of plans in the memoization table that represents the optimal plan. The root plan (satisfying the *query goal*) will reference one or more sub-plans, which again may reference more sub-plans. The reconstruction phase rebuilds the operator graph from this plan hierarchy.

The plan generator itself is not involved in this step — it is left up to the rules to enable them to do whatever they want during this phase. This promotes extensibility. IRule, which all constructive rules implement, offers the *BuildAlgebra* method, which is responsible for building the operator node for itself. Each plan in the plan hierarchy was also tagged with the rule that produced it during the search phase. The plan generator starts the reconstruction phase by calling *BuildAlgebra* on the rule that produced the root plan, passing the plan to it. This rule is again responsible for calling *BuildAlgebra* on the rules producing its input plans. Since the plan hierarchy may form a DAG, each rule may be asked to construct the same operator node twice. Therefore, during reconstruction, a *Reconstruction table* is used to just return the previously constructed operator node. Finally, the completed operator node graph is returned as the planned query.

Listing 3.14 shows the *BuildAlgebra* method for SelectionRule.

3.6.7 Improvements And Additions to Neumann's Design

Multi queries. We have extended the design to handle multi-queries, including possible construction of DAG-plans and globally optimal costing for multi-query.

Rewrite steps / Transformation rules. We have added transformation rules to enable pre-/post-processing and query rewriting. This takes the solution from a plan generator to a full optimizer. See Section 3.5.

Adaption of MARS' algebra. We have adopted a design that was intended for a database to a search engine with an unusually rich algebra.

Base plan sharing. By using the algorithms described in [Neu05], base plans (the leave nodes, such as scans and index lookups) will not be shared as they are entered into the memoization table under their own produced properties.

We have made base plan sharing possible by entering share equivalent representatives into the memoization table for all their share equivalent rules as well.

Left-deep join enumeration. The implementation of joins in [Neu05] and [NM08] only considers bushy join plans. To show that the design is extensible, we wanted to implement left-deep plan enumeration. This is currently working and (unsurprisingly) performs better in terms of time spent optimizing, but will result in suboptimal plans. See explanation in Section 5.3.9

Dynamic rule discovery. We have added the rule binder mechanism for dynamic rule loading by the optimizer, as described in Section 5.1.

Caching of unreachable plans. Often, the optimizer is asked to produce a plan that is not possible to create. In the referenced works, it is suggested that the `Filter` property should always be used to find such cases. We propose to cache the result of the filter check in the memoization table for better performance. Quick performance tests indicate that for a query with 9 relations, enumerating bushy plans, this yields a **performance improvement from 2.15s to 0.93s**. See explanation of Listing 3.10.

Nicer implementation of BitSet (property set). We have implemented a property set that behaves like a set of strings (nicer to work with and debug), but still performs well and offer compact bit mask storage. This is described in Section 3.6.1.

Visualization of query plans. We have implemented query plan visualization. The result can be seen in Chapter 7.

Managed implementation in C#. Our implementation is in C# — a managed language, focusing on an extensible and clear implementation, utilizing features such as attributes and interfaces.

Refactored interfaces. Some of the rule interfaces have been refactored (e.g. IProducerRule) to allow for cleaner implementation. See description under Section 5.3.1.

We also have more improvements we have not had time to implement yet, see Section 8.3 and Section 9.1.

3.7 Share Equivalence and DAG Construction

So far, we have only slightly touched how DAG query plans are created. We now explain how this happens, from concept to implementation.

3.7.1 Sharing and Share Equivalence

As the optimizer in [Neu05], our optimizer constructs *physical* DAGs. This means that it allows operators to produce multiple logical query expressions simultaneously. For instance, for the query in Figure 1.8a on page 18, the two joins between B and C are different logical expressions since they have different variable bindings to B and C. However, in Figure 1.8b the join between B and C is shared and is producing both logical expressions. This can be done since they differ only on variable bindings — they produce exactly the same result.

The plan generator must be able to recognize such possibilities for sharing, and must at all times be on the lookout for plans that represents the same logical expressions and can be shared. To be able to recognize logical expressions that are equivalent, the concept **share equivalence** is

introduced. Two expressions are **share equivalent** *if one expression can be computed by using the other expression and renaming the result* [Neu05]. Renaming the result here refers to changing the variable bindings.

Share equivalent expressions can range from entire queries to the smallest of sub-plans, and can be used to share partial results at all levels. For example, consider the query `SELECT * FROM tree AS node JOIN tree as parent ON (node.parent_id=parent.node_id)` — a self-join of tree. This is a trivial example of how the scan of `tree` — a leaf node in the plan — could be shared.

Another opportunity for DAG generation, one we have not had time to study, is adding compensating operators when the input cannot be completely shared, but is close enough that large parts can be reused. We give an example in the next subsection.

3.7.2 Share Equivalence for MARS's Operators

We need a rule set to decide what operator expressions are share equivalent, and we now quickly cover what conditions must hold for expressions constructed with MARS' operators to be share equivalent, as shown in Table 3.2. $\delta_{A,B}$ means the renaming function for attributes from A to B. $\mathcal{G}(\vec{A})$ is the grouping fields of A, and $\mathcal{A}(\vec{a})$ is the aggregates.

Lookups provide the foundation as they are share equivalent if they look up the same word in the same index. The remaining operators require share equivalent inputs and some additional requirements: ScoreOccurrences, ONear/Near and Trim require equally set properties. Map and Select require that the input attributes is mappable, while joins require that the join predicates

Lookup$(index_1, word_1)$	$\equiv_S$	Lookup$(index_2, word_2)$
iff $index_1 = index_2 \land word_1 = word_2$		
ScoreOccurrences$(A_1..A_n, properties_1)$	$\equiv_S$	ScoreOccurrences$(B_1..B_n, properties_2)$
iff $A_{1..n} \equiv_S B_{1..n} \land properties_1 = properties_2$		
ONear/Near$(A_1..A_n, properties_1)$	$\equiv_S$	ONear/Near$(B_1..B_n, properties_1)$
iff $A_{1..n} \equiv_S B_{1..n} \land properties_1 = properties_2$		
Map$(A, a = f(b))$	$\equiv_S$	Map$(B, d = g(c))$
iff $A \equiv_S B \land \delta_{A,B}(b) = c \land f \equiv g$		
Select$_{a=b}(A)$	$\equiv_S$	Select$_{c=d}(B)$
iff $A \equiv_S B \land \delta_{A,B}(a) = c \land \delta_{A,B}(b) = d$		
Join$(A, B, \vec{a} = \vec{b})$	$\equiv_S$	Join$(C, D, \vec{c} = \vec{d})$
iff $A \equiv_S C \land B \equiv_S D \land \delta_{A,B}(a_{1..n}) = c_{1..n} \land \delta_{A,B}(b_{1..n}) = d_{1..n}$		
Group$(A, \mathcal{G}(\vec{A}), \mathcal{A}(\vec{a}))$	$\equiv_S$	Group$(B, \mathcal{G}(\vec{B}), \mathcal{A}(\vec{b}))$
iff $A \equiv_S B \land \delta_{A,B}(\mathcal{G}(A)_{1..n}) = \mathcal{G}(B)_{1..n} \land \delta_{A,B}(a_{1..n}) = b_{1..n}$		
Trim$(A, \text{offset}_1, \text{hits}_1)$	$\equiv_S$	Trim$(B, \text{offset}_2, \text{hits}_2)$
iff $A \equiv_S B \land \text{offset}_1 = \text{offset}_2 \land \text{hits}_1 = \text{hits}_2$		

Table 3.2: Share equivalences for MARS' operators

```
1  bool ShareEquivalentRules(IProducerRule r1, IProducerRule r2) {
2     if (!r1 and r2 are structurally identical)
3        return false;
4     if (r1 is IBaseRule && r2 is IBaseRule)
5        return true;
6     reqRules1 = {r | r ∈ rules ∧ r.produced ⊆ r1.required}
7     reqRules2 = {r | r ∈ rules ∧ r.produced ⊆ r2.required}
8     if (!∀ i ∈ reqRules1 ∃ j ∈ reqRules2 : ShareEquivalentRules(i, j) ||
9        !∀ i ∈ reqRules2 ∃ j ∈ reqRules1 : ShareEquivalentRules(i, j))
10       return false;
11    return true;
12 }
```

Listing 3.15: *ShareEquivalentRules*, pseudo-code

are mappable. Group requires that the grouping fields are mappable, the aggregates the same and that the aggregate input is mappable.

For example, two expressions with a Select on top of a Lookup are share equivalent if the selection predicates are equal (with attribute renaming taken into account) and the lookups are share equivalent.

As a trivial example of how compensating operators can help, consider the requirements for share equivalence between two Trim-operators with otherwise share equivalent inputs. If one limits the number of hits to 20 and the other limits it to 10, they are *not* share equivalent with the requirements of Table 3.2. However, for non-trivial plans, reducing 20 results to 10 is evidently cheaper than evaluating the entire sub-plan twice.

In practice, the plan generator uses the algorithm given in Listing 3.15 to compare two rules for share equivalency. We have partially reverted to pseudo-code for the algorithms in this section to save space. Here, being *structurally identical* means that they have the same properties, joins the same join keys and so on. Structurally identical base rules are trivially share equivalent. The last part checks whether there exits share equivalent inputs from r1 for r2 and vice versa.

3.7.3 Equivalence Class Construction

During the preparation phase, after all the rules for the query have been instantiated, the plan generator analyzes the rule set to find share equivalent optimization rules using the previous algorithm. As shown in Listing 3.16, it will iterate through all rules topologically (this is needed because of the transitive attribute mapping process) and gather share equivalent rules in equivalence classes. One representative is chosen for each class, and the EQClasses dictionary maps the class representative to the class. It will also call *MapShareEquivalentAttributes* on each rule to gather mappings for share equivalent attributes produced by the rules. Then, equivalence classes with only one member are removed, as no sharing can happen here.

Finally, a ShareEquivalenceMap is generated, mapping the produced bit properties of all members in a class to the produced properties of the class representative.

For example, the query in Figure 3.5a has 10 operators, L_0, L_1, L_2, L_3, L_4, $L_0 \bowtie_5 L_1$, $L_3 \bowtie_6 L_4$, $L_1 \bowtie_7 L_2$, G_8, where L means Lookup, $\bowtie$ Join and G Group. Intuitively, the lookups in the bottom are pairwise share equivalent since they lookup the same word, hence $L_0 \equiv_S L_3$ and $L_1 \equiv_S L_4$. The joins $\bowtie_5$ and $\bowtie_6$ have the same join predicate and joins share equivalent inputs, and hence they are share equivalent: $\bowtie_5 \equiv_S \bowtie_6$. We have three equivalence classes, and we assume from now on that L_0, L_1 and $\bowtie_5$ was chosen as representatives.

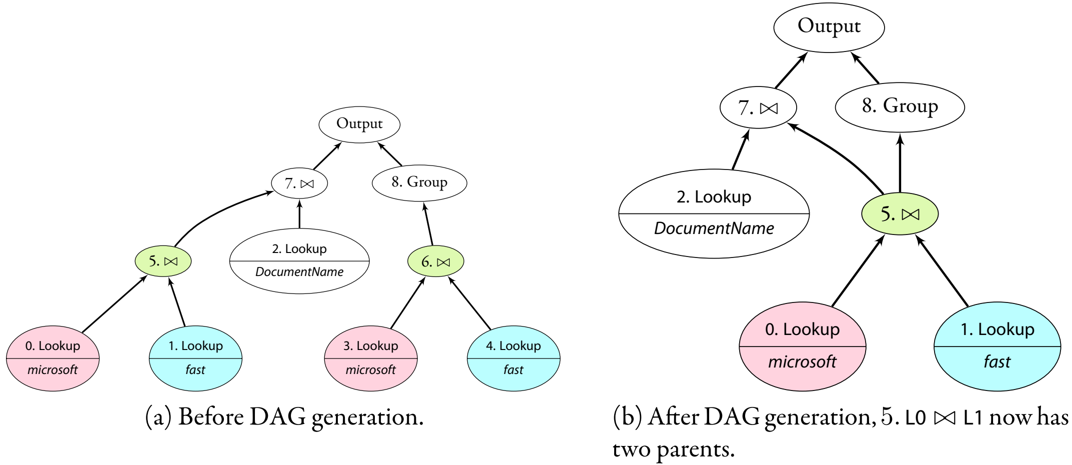
```
void ConstructShareEquivalenceClasses() {
   EQClasses = new Dictionary<IProducerRule, HashSet<IProducerRule>>();
   EQAttributeMappings = new Dictionary<NodeAttribute, NodeAttribute>();
   foreach(rule ∈ GetRulesTopologicallySorted()) {
      if (∃ (a,b) ∈ EQClasses : ShareEquivalentRules(a, rule)) {
         b.Add(rule)
         a.MapShareEquivalentAttributes(rule, EQAttributeMappings);
      } else {
         EQClasses.Add(rule, new HashSet() { rule });
         rule.MapShareEquivalentAttributes(rule, EQAttributeMappings);
      }
   }
   foreach ((a, b) ∈ EQClasses : |b| == 1)
      EQClasses.Remove(a);
   ShareEquivalenceMap = new Dictionary<BitSet, BitSet>();
   foreach ((a, b) ∈ EQClasses) {
      ShareEqRepr |= a.Produced;
      foreach (rule ∈ b)
         ShareEquivalenceMap.Add(rule.Produced, a.Produced);
   }
}
```

Listing 3.16: Equivalence class construction, pseudo-code

3.7.4 Goal Rewrite

During plan generation, every time the plan generator is asked to build a goal, it will try to rewrite the goal to use equivalence class representatives. We saw this happen as part of *GeneratePlans* in Section 3.6.5. From Figure 3.5a, if it is asked to produce $L_3 \bowtie_6 L_4$ which is share equivalent with $L_0 \bowtie_5 L_1$ (and the latter are the representatives), it will rewrite the goal from $L_3, L_4, \bowtie_6$ to $L_0, L_1, \bowtie_5$. This new rewritten goal is then used (in addition to the original goal) to produce plans.

It is the method *ShareEquivalentGoal* in Listing 3.17 that performs the rewrite. Basically, it tries to apply each mapping in the previously constructed `ShareEquivalenceMap` to the goal. If the entire *from* part is present in the goal, *from* is replaced by *to*. Lastly, it checks whether it was able to perform a complete rewrite (with the equivalence class representatives taken out, since they would be rewritten to themselves), as the plan generator will only search for plans



(a) Before DAG generation.

(b) After DAG generation, 5. $L0 \bowtie L1$ now has two parents.

Figure 3.5: DAG creation through goal rewriting

```
1 public bool ShareEquivalentGoal(BitSet goal, out BitSet sharedGoal) {
2    BitSet sharedGoal = goal;
3    foreach ((from, to) ∈ ShareEquivalenceMap)
4       if ((sharedGoal ∩ from) == from)
5          sharedGoal = sharedGoal \ from ∪ to;
6    return sharedGoal ≠ goal ∧ sharedGoal ∩ (goal \ ShareEqRepr) ≠ ∅;
7 }
```

Listing 3.17: Equivalence class goal rewriting, pseudo-code

```
1 plan.Sharing = left.Sharing ∪ right.Sharing;
2 if (left.Shared ∧ right.Shared ∧ This rule is eq. class representative) {
3    plan.Sharing ∪= this;
4    plan.Shared = true;
5 }
```

Listing 3.18: Setting `Sharing` and `Shared` properties for plans, pseudo-code

using this new goal if it is a complete rewrite. If it is not, it means that not the complete sub-problem is share equivalent, only that a smaller sub-problem is. In that case, a rewrite will be tried and succeed later, when the plan generator is working on the smaller sub-problem that is share equivalent. See Section B.3 for the full source code.

To see how this generates DAGs, consider the example with the two joins above which exist in different parts of the same query. At one point, the plan generator will produce a plan for $L_0 \bowtie_5 L_1$ when working on this part of the query. The plan will be memoized with the properties L_0, L_1, $\bowtie_5$. Then, when it is later asked to produce a plan for L_3, L_4, $\bowtie_6$, this goal will be rewritten to L_0, L_1, $\bowtie_5$ and searched for. This will immediately yield a cache hit in the memoization table and will return *the same plan* as for $L_0 \bowtie_5 L_1$. The result is that the plan for $L_0 \bowtie_5 L_1$ now has two parent plans as can be seen in Figure 3.5b, and we have an in-memory DAG which is later reconstructed to a DAG query evaluation plan.

Some care must be shown for base plans, however, since for example a search for L_3 will immediately yield a cache hit and not trigger the rewrite procedure. Instead, rewrites are carried out when the base plans are entered into the memoization table during the preparation phase.

3.7.5 Sharing Properties for Plans

We touched this when presenting the data structure for a plan, but we will now explain it more thoroughly. The `Sharing` bit set for a plan stores how much of the plan is potentially shareable. More specifically, each equivalence class representative will be assigned a bit in this bit set, which will be set if the representative is shareable.

For example, in the previous example, the plan generator would set the sharing bit L_0 for the plan L_0, but not for the plan L_3. The plan $L_0 \bowtie_5 L_1$ would have sharing bits for L_0, L_1 and $\bowtie_5$. However, the plan $L_0 \bowtie_5 (L_1 \bowtie_7 L_2)$ would only have sharing bits for L_0 and L_1 as $\bowtie_5$ cannot be shared. This is because only whole sub-problems can be shared — as the output from $\bowtie_5$ also includes $\bowtie_7$ in this case and cannot replace a share equivalent problem to $\bowtie_5$.

To detect this last case, the `Shared` property is used. It says whether this exact plan is shareable. Whenever a plan is constructed, two conditions must be satisfied for it to be shareable: 1) The rule constructing it is the equivalence class representative, and 2) all inputs have `Shared` set to *true*. This is shown in Listing 3.18.

Storing sharing opportunities with the plans has two purposes. First, it is used when pruning plans, as a plan only dominates another if it is cheaper and offers at least as many sharing

opportunities and orderings. This can be determined by checking if the dominated plan's Sharing property is a subset of the dominator's. Second, the cost model needs to know how much of a plan is shared to correctly estimate the cost of it, as n reads of a plan do not imply n times the cost if it is shared. It switches to a DAG-aware cost calculation if the Sharing properties of the input plans for a rule overlaps. Also, if a plan that can be shared n times cost more than n times another optimal plan, it can be pruned away.

3.8 Graph Pattern Matching

Graph pattern matching is the procedure of finding a sub-graph in a graph that matches a given pattern — such as *"a MergeJoinOperator with at least one MergeJoinOperator child"*. We use this to declare patterns that trigger rules that transform the operator graph. The example pattern (although in natural language) is used to merge adjacent MergeJoinOperators — after also checking that the join predicate is the same, but this is left out of the pattern to keep it simple. We also use it in rule binders for constructive rules, which usually look for single operator nodes to instantiate rules for. Rule binders are used for constructive rule initialization during the preparation phase of plan generation, and is dealt with in Section 5.3.1. Patterns make it easy for implementers of rules to *declaratively* express what they are looking for, instead of having to code the search themselves. In Section 3.9 (Compound Rules) we elaborate why rule binders have patterns and not a simple one-to-one-mapping between operator types and rule binders: Often, there are certain usage patterns of operators which can be grouped together to a single search rule, which in turn reduces the search space.

Secondly, separating concerns is generally a good practice. Traversing and finding subgraphs are tasks we do in many parts of the optimizer. Subtle bugs are exposed easier if the same code is exercised in many different ways than if every component matches the graph its own way.

Lastly, expressing patterns declaratively provides an avenue for future optimization. The authors are no experts with state machines or regular languages, but we believe it is possible to combine several patterns into a single state machine, which can greatly reduce the amount of graph traversal currently done. However, we have not focused on these kinds of optimizations, since it is not part of the search phase of the optimization. For example, [CGK05] describes an algorithm that runs in polynomial time in directed acyclic graphs, while [Gei08] is a full solution for graph rewriting — where users define before- and after patterns the graph is then rewritten to. The latter has a GPL-license, which makes it problematic to use if *fast* is ever to use any of our code, which of course we hope they will do. Instead, we implemented a very simple matcher. There is only a before pattern, which then triggers a callback in the rule which gets to rewrite the graph. Our Node-class has sufficient functionality to make altering the graph easy.

3.8.1 Implementation

We only briefly describe how our naïve pattern matcher is implemented, as it is not an integral part of our thesis.

Patterns are expressed by constructing a graph consisting of AbstractNodeMatchers. AbstractNodeMatcher is an abstract base class for different node matchers, each used to match a node of some kind. It is also a node — which means we can perform pattern matching on patterns!

AbstractNodeMatcher has a simple interface. Most important are the HasMatched-property and the *Search(Node node)*-method. HasMatched returns whether the matcher is in a matching

state — i.e. the nodes input so far satisfy the pattern. *Search(...)* accepts a node and returns whether it matches — i.e. whether it *keeps* the matcher in a matching state. The statefulness is needed by matchers where a match is a function of all nodes provided. Thus the need for both *Search* and `HasMatched`.

The most interesting part of the matchers, however, are the traversers. We have an AbstractTraverser whose subclasses simply return an iterator given an input node — such as ascendants or descendants in breadth first, topological ordering, etc. Given an iterator, the *Search(...)*-method uses the `RootMatcher`-, `TakeWhile` and `StopPattern`-sub-patterns. The `RootMatcher` is a pattern that defines where traversal will start. For every node returned by the `RootMatcher`, we start traversal. Whether we traverse the roots ascendants or descendants depend on the specific AbstractTraverser-subclass. Traversal continues until the `StopPattern` (if any) matches and/or as long as the `TakeWhile`-pattern (if any) matches.

With these simple primitives, we have implemented the following matchers:

NodeTypeMatcher matches an operator node of a given type, for example SelectOperator or LookupOperator.

NodeBehaviorMatcher matches an operator node satisfying a specified behavior. The defined behaviors are:

SetPreserving: operator is not allowed to remove or add tuples to the set.

DataPreserving: operator is not allowed to alter the tuple data.

OrderPreserving: operator is not allowed to alter the order of the tuples.

ExpressionMatcher allows for specifying an arbitrary delegate (`Node` → `bool`) for matching a node. LINQ (Language Integrated Query) compiled expressions are used for performance.

AnythingMatcher: matches any node — i.e. always returns true. Similarly, NothingMatcher always returns false.

AllMatcher: takes a sub-pattern and returns true until an input does not match, after which it always returns false. Similarly, AnyMatcher returns true if any input has matched.

AndMatcher: takes multiple sub-patterns and returns true for nodes that matches all patterns. Conversely, OrMatcher returns true if any sub-pattern matches.

GroupMatcher: wraps a sub-pattern to name its set of matches, much like groups in regular expressions.

RangeQuantifier: returns true when at least min and at most max nodes have matched the sub-pattern.

ChildTraverser: Takes `RootMatcher`-, `TakeWhile` and `StopPattern`-sub-patterns, and traverses the children of all roots, as long as `TakeWhile` matches and until `StopPattern` has matched.

With some sugar, we can then construct the pattern in Listing 3.19. It matches join operators with at least one join as input. The example is from a post-processor which considers whether adjacent merge joins should be combined into a single join in a multi-join. The equivalent pattern constructed with actual operators is shown in Listing 3.20.

AbstractNodeMatchers are usually not used directly — instead, a GraphSearcher is created with an input pattern. First it traverses the *pattern* looking for any MatchGroupers, creating a

```
Match.NodeOfType("MergeJoinOperator").GroupAs("join")
.WithAllChildren(
    Match.AtLeast(1, Match.NodeOfType("MergeJoinOperator"))
);
```

Listing 3.19: Sample Graph Pattern

```

new ChildTraverser() {
  RootMatcher = new MatchGrouper("join", new NodeTypeMatcher("MergeJoinOperator")),
  TakeWhile = new AllMatcher(new RangeMatcher(new
  NodeTypeMatcher("MergeJoinOperator", 1, -1)))
}
```

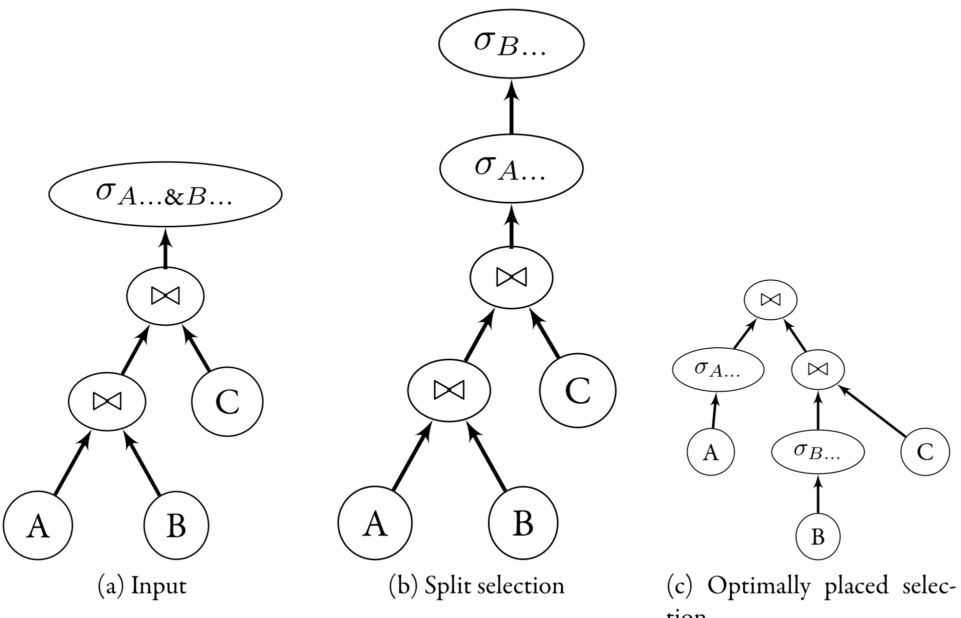
Listing 3.20: Operators of Sample Graph Pattern

dictionary mapping the name of the group to the list of nodes matched by the wrapped pattern. Then, it traverses the input root node top-down (breadth first). For every node visited, the pattern is attempted matched. If found, the node is added to the list of the searcher's matched nodes. Also, the list of matches for the various groups are updated. If a match has been found, the GraphSearcher's *Search(...)*-method returns true, and the matches are available in the object.

3.9 Compound Rules

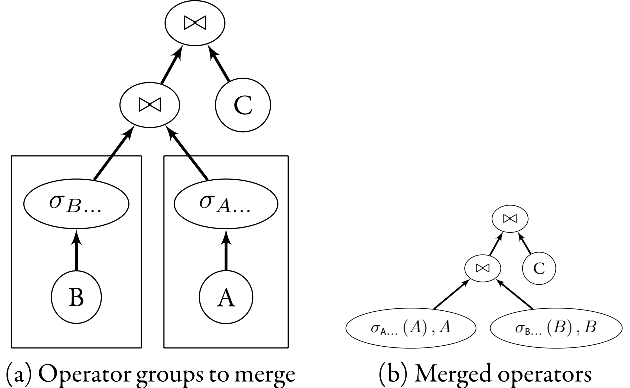
The size of the search space grows exponentially with the number of nodes in the query graph. Oftentimes, there are certain combinations (patterns) of operators that are not worthwhile to try and reorder. One such example is an (O)Near-operator above Lookup-operators, as well as a ScoreOccurrences-operator above one or more Lookup- and/or (O)Near-operators. With the example query shown in Figure 1.2 (page 7) we can coalesce the sub-graph with ScoreOccurrences as the root to a single compound node. The impact is a greatly reduced search space: 11 rules are reduced to 7. (Remember that rules are not instantiated for Output-, Copy- and Sort-operators.) In this case, however, ScoreOccurrences and the sub-graph below it is quite constrained and will not be attempted moved around too much anyways. However, imagine the input shown in Figure 3.6, with a single selection above a join whose predicate references two relations. The desired plan is shown in Figure 3.6c. To be able to push down the predicates, we need a pre-processor to split the Select-operator into two — which we can do, because the predicates are AND-ed together. However, by doing so we drastically increase the search space, as Select-operators are not as "well-behaved" as the sub-graph contained by the ScoreOccurrences. This is because few operators *depend* on the Selects. In this case, only the omitted Output-operator would depend on the Selects. The consequence is that a lot of permutations of Select-placements are attempted: for every join combination and for every possible placement of $\sigma_{A...}$, every possible placement of $\sigma_{B...}$ is considered. If, however, we can reason that it is likely that the best placement of the Selects are immediately above the input they depend on, we can merge the operators into one.

For the join ordering shown in Figure 3.6b, there are $4 + 4 + 3 = 11$ possible placements of the Select-operators for the orderings $(A \bowtie B) \bowtie C$ and $(B \bowtie A) \bowtie C$. In addition, there are four other possible join orderings (two with A last, as shown, and two with B last), each with $4 + 3 = 7$ possibilities. This gives us $2 * 11 + 4 * 7 = 50$ possible plans. With compound operators as shown in Figure 3.7b there are just 6 possible plans. The savings are of course of larger magnitudes with more complex queries, especially in combination with other



(a) Input (b) Split selection (c) Optimally placed selection

Figure 3.6: Splitting selections



(a) Operator groups to merge (b) Merged operators

Figure 3.7: Merging selections

operators with loose dependencies.

However, when dealing with compound rules, one must be careful with respect to shareable operators in the sub-graph of the compound rule. Since the whole point of the compound rules is to reduce the search space, shareability of contained operators will not be considered. Thus, we must be careful not to mask away important optimization opportunities.

3.10 Integration with MARS

To get MARS to optimize the queries, we have injected a component into the query processing pipeline. This is done by creating the “dynamic component” OptimizerTransformer, which is then run-time-configured to be invoked before query evaluation. Figure 3.8 shows how the optimizer fits in.

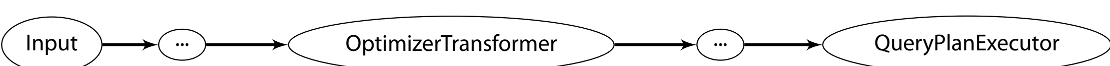


Figure 3.8: MARS’ evaluation pipeline

A lot of infrastructure is needed to have the component created, configured and injected correctly — leading to daunting class names such as OptimizerTransformerOperatorBuilderSource. However, the most important part is the OptimizerFacade.

OptimizerFacade knows how to convert from the graph structure in MARS to the one used by our optimizer — and vice versa. In addition to checking if the query conforms to the form we expect it in, that is all the facade does: Convert, optimize, convert back and replace the query to be executed.

Conversion *from* the MARS node involves instantiating an OperatorNode, setting the operator type, and reflecting all attributes. We walk the graph topologically, to add the relationships by adding children to the parent.

Conversion *to* MARS nodes is done by invoking MARS' operator factory. We had to reflect our way around some visibility issues, and get the factory injected through all the infrastructure separating the facade from the processing chain. With that done, the conversion is simple.

4 Cost Estimation and Statistics

"If you don't find it in the index, look very carefully through the entire catalog."

— Sears, Roebuck, and Co., Consumers' Guide, 1897

4.1 Introduction

The goal of query optimization is to find the *best* plan (or one that is reasonable close to it), which is the *cheapest* by some cost-metric. But how do we define "cheapest", and how can we be reasonably sure the cost estimates are correct? Also, the definition of cheapest may wary. Often, the goal is to complete the query in the shortest time possible, but this is not always the case. Other goals may be to minimize the time used to return the first records — this is reasonable in a search engine setting, where often only the first few results are interesting. In situations where the system is getting congested, the goal may be to reduce the global average response time, for example by reducing resource usage per query to increase concurrency. Furthermore, if the query is being executed on several nodes, communication costs might need to be minimized, not only due to their latency, but also because the network connections can become congested. Changing network links and node availability and load are a few examples of environmental changes that should affect the plan cost calculations.

The focus of this master's thesis has not been to accurately model the costs of MARS' operators and keeping up with environmental changes , but rather to create a framework that can be used to implement such a model later. MARS is closed-source, and we have very limited knowledge of the detailed implementations of the different operators, which is required if we were to cost model them in detail. To be able to test the framework, we have therefore implemented a simple cost model that takes things like tuple sizes and cardinality into consideration, using simple models for the operators we have looked at. By simple, we mean "school book" models that may not be accurate for MARS, specifically, but sophisticated enough to, for instance, prefer merge joins over hash joins and DAG-plans over non-DAG-plans — or vice versa. When creating the framework, we made sure to get an overview of the most important aspects and to take informed design decisions that do not rule out necessary functionality to get the cost- and statistics modeling better later on.

The most important part is achieving a loose coupling between the optimizer, the rules and how the costs are modeled and eventually calculated. However, no matter how crude the cost modeling is, we must also consider the fact that when dealing with DAG-plans, the local optima may not necessarily be the global — since a sub-plan could possibly be shared by many parent operators. Also, it is worth noting that cost modeling is only performed during the *plan generation* or *search phase* of query optimization, not during pre- and post-processing.

In Section 4.2 we describe what influences the costs, and partially why it is important to

abstract the calculations. Section 4.3 shows some example uses of statistics and how it influences the planning process, with Section 4.3.2 covering the lack of statistics in MARS. Then, Section 4.4 describes why a cost *component* makes sense, and how it works.

4.2 Cost Factors

To calculate a plan's costs, several factors are considered. Available buffer space, amount of I/O needed to fetch the necessary pages, the probability of finding the page in the operating system's page cache, sequential vs. random reads, etc. are a few examples of factors that affect costs.

An operator, given expected input sizes and selectivities of the predicate it applies, can give a reasonable estimate of the resources and time it needs to do its task. It can estimate the buffer sizes and the amount of memory it needs, expected random and sequential reads and the expected size of the temporary relations needed if operations spill to disk. However, it does not make sense to express this as an arbitrary number "cost" from the operator's viewpoint. For example, if a conventional disk is replaced with a solid state drive that provides cheaper random reads, the cost is reduced even though the resource requirement remains the same for the operator. It is not the operators individual tasks to determine the actual cost of its requested operations. We leave that to a cost manager, which translates the needs of the operators into a cost which can be used to compare plans.

To do this properly, we need to know what constitute the costs of an operation:

- **I/O** contributes with a lot of the cost involved in evaluation of queries. Data need to be read from somewhere — be it hard drives or page cache — and written somewhere. Moreover, with conventional hard drives there is also a substantial difference in the time it takes to perform sequential vs. random reads.

  If the data is too large to fit in memory, the intermediate result sets may need to be flushed to disk and re-read several times during evaluation ...

- ... as **memory** is a limiting factor. In a concurrent environment, memory usage per client is typically limited to prevent a few users from spending all the memory. If the result set is just a single tuple more than can be held in the available (or permitted) memory and the result set is to be sorted or joined, external sorting must be used [Bra03].

- **CPU**: *"A well-tuned database installation is typically not I/O-bound."* [SH05a] When the right mix of I/O-subsystems and memory is available, I/O latencies are no longer the bottleneck. Stonebraker et al. points out that in such a system, memory copies are becoming the dominant bottleneck, due to the gap in performance evolution between CPU cycles and RAM access speed. However, not all systems have the luxury of having abundant memory and quality I/O-subsystems.

  Furthermore, the memory budgets can differ with light and heavy loads. With no queries in the admittance queue, the optimizer can decide to be generous when handing out memory. With heavy load, memory constraints might be tightened.

- **Communication costs** are a limiting factor in distributed environments. For example, if joins are to be evaluated with data from several nodes, the optimal join ordering can possibly be the one that minimizes the amount of network traffic needed to transfer the intermediate results.

As we will see, in our cost model, we have modeled the cost as a linear combination of these factors in two floating point numbers. In addition to the cost factors given above, *ordering* and *sharing* also influences which plans are *dominated* by other plans. For a plan to be dominated, there must exist another plan that is logically equivalent (produces the same properties) that is equal or better for both costs, ordering and sharing. The optimizer compares all logical equivalent plans as soon as they are created and immediately prunes all plans that are dominated. For *costs*, lower is better, while for *ordering*, a more specific ordering is better. For instance, a plan that produces output ordered by (a,b,c) is better than one that produces (a,b), all else being equal. For *sharing*, a plan that offers to share more (but does not necessarily do it *yet*) is better.

4.3 Importance of Statistics

Repeating the example from Section 1.6.2, we have the following query:

$$\sigma_{\text{person.birth}>1950-01-01}\left(\text{person} \bowtie \sigma_{\text{city.name}=\text{Å}}\left(\text{city}\right)\right) \quad (4.1)$$

We argued that the amount of people born after 1950-01-01 in Norway is larger than that of the small village — an important realization when deciding how to join. To be able to guesstimate these cardinalities and selectivities and thus the costs, the optimizer must consult statistics about the various relations. Without them, the optimizer cannot reason about the costs, which would make large parts of the optimization process moot. In Section 2.2, we described how System R estimates cardinalities and selectivities — and the limitations with its approach. Since then, a lot of effort have gone into determining how to devise and use statistics to estimate these — without being too expensive to use or maintain. In most cases, selectivity estimates directly affect the decision of what the cheapest plan is, so it is important to be as accurate as possible [CR94].

Some examples of statistics used are:

- **Cardinality** of the relation — e.g. the number of records in the table/index. For example that there are 4.8 million records in the person-table.
- The on-disk **page count** of the relation.
- **Histograms** defining the number of records within various value *ranges*. For example the number of people within different post code ranges.
- **Most common values**. For example that there are more people living in some of the post codes than the histograms can express.
- The number of **distinct** values.
- Average **tuple size**.
- Correlation **logical vs. physical ordering** on the disk, which can express the degree of random reads needed.

Most of these are relevant to MARS, except maybe the last, since often the indexes are clustered (i.e. the on-disk ordering of records equals the logical one) on the most commonly used join keys anyway.

When estimating the selectivities, it would be preferable to not necessarily assume independence of the predicates. For example, while the terms "query" and "optimization" are both uncommon, the sentence "query optimization" is even less common — except maybe in this

report. So estimating the selectivity as $\mathcal{S} = \mathcal{S}_{query} \times \mathcal{S}_{optimization}$ is inaccurate. Another example is "in" and "the" — two very common words, where the sentence "in the" is quite common as well. Thirdly, "the" and "who" are quite common, but "The Who" is rare. With collocations available, better estimates can be used. How to devise these is outside the scope of this thesis, however. See [MS99, Chapter 5] for more information on collocations. While creating them can be quite laborious, looking them up during optimization is fairly cheap.

In the report of our specialization project [BN08], we included a fairly complete example of how lists of the most common values and data range histograms were used to estimate selectivity. However, since we lowered the priority of accurate cost modelling, we have not studied it further nor implemented it, so we omit the example in this report. The example also illustrated the computational costs involved with statistics. For every predicate on every relation, histograms and lists of most common values are involved in estimating selectivities. By increasing the size of the histograms and list of most common values, the accuracy of the estimates, the computational effort, space- and maintenance cost increases. The challenge is finding a sweet spot where the performance advantages do not severely offset the costs.

4.3.1 Statistics Gathering

The main problems with statistics are gathering and keeping them up to date. These are issues we have not had time to look deeply into, but we have found one interesting approach we find worth mentioning.

The most common technique involves regular random sampling [CMN98] — as reading the entire relations quickly or keeping them completely up to date become costly. This has been studied extensively and has not got too much to do with query optimization.

However, in [CR94], Chen and Roussopoulos proposes an interesting approach where *"The real attribute value distribution is adaptively approximated by a curve-fitting function using a query feedback mechanism."* This approach is interesting, because it is less prone to choosing bad random samples, and more likely to converge to fairly accurate estimates on common queries. Without having studied this in depth, we believe that this would contribute unacceptable overhead if done on every query. Thus, it could be the task of an optimizer to schedule when and where such operators should be added.

4.3.2 Statistics and MARS

MARS does not currently store any statistics usable in an optimization context. As such, we are in a position where we can suggest what statistics should be made, and how they should be gathered. However, this has been outside the scope of this thesis — partially because we have not been provided with realistic data-, index- and query sets. Consequently, it is difficult to suggest anything else than general purpose approaches. Moreover, we have focused our efforts on other aspects, as explained earlier.

However, to be able to do any estimation at all, we need *some* statistics. We had to stub the system catalog to provide us with two basic statistics for a specified index: 1) The number of occurrences per word (which we generated by processing the files we fed to the indexer), and 2) the average tuple size. This is the absolute minimum needed to make any sensible estimates on operator costs and selectivities and we suggest that MARS in the future at least implements these. Another statistic that would be useful to have is the number of documents per word, which would allow better estimates for operators succeeding ONear/Near and ScoreOccurrences.

```
public enum CostRelation {
   Better, Worse, Equal, Unknown
}

public interface ICost {
   CostRelation Compare(ICost other);
   CostRelation CompareTotal(ICost other);
}
```

Listing 4.1: Plan generator's cost model interface.

4.4 Cost Component

Much in the same way the optimization rules are external to the optimizer, the logic that has to do with plan costs are located in the *cost component* which is external to the optimizer. In that way, we achieve a clean separation between the optimizer core and cost modeling. This is beneficial if we were to switch out the current simple cost model with a more advanced one in the future. We would be able to do this without too many changes (ideally none) to the optimizer core itself.

In the following sections, we look at the cost component from three viewpoints: 1) As seen from the *plan generator* (part of the core) — ultimately, all the optimizer core cares about is *if* plan A dominates plan B — not *why*. 2) As seen from the *Optimization rules* — the rules need access to some cost primitives when expressing the costs of the plans they produce. The cost model for the individual rules can be found later in this chapter. And finally, 3) how *cost state information* is stored together with the plans, i.e. how the costs of a plan is described.

4.4.1 Plan Generator Interface

During optimization, the plan generator will generate possibly millions of plans, of which many perform the same logical operations, but have different costs, orderings and sharing opportunities. As Section 3.6.3 explains, the PlanSet structure is responsible for removing dominated plans. To do this, it needs to be able to compare plans by costs. It does not care *why* a plan is cheaper than another, just *if* it is. This makes the rules simpler, and the cost estimation more flexible. For example, it can be aware of environmental changes, as mentioned in the introduction. Listing 4.1 shows the interface the optimizer expects a plan's costs to implement. It allows the plan generator to compare two arbitrary plans by costs without knowing the details involved. Practically, to switch to another cost model, what would need to be done is to change the rules to use the new model and make it implement the same ICost interface.

CostRelation's Unknown-member and ICost's *CompareTotal* warrant an explanation. When the cost component cannot decide if a plan is always better than another, for example because it uses less CPU but more I/O, it returns unknown. This means that it may not exist a total ordering between the plans with regards to costs. The decision is then left to the planner whether or not to keep the plan. In most cases, the planner will keep both plans and decide between them at a later stage during plan generation.

When the planner calls *CompareTotal*, however, the cost component *cannot* return unknown. This is requested at the end of the planning phase, when the planner has a small amount of plans it needs to choose one from. For example, in the search phase, *Compare* may return Unknown because the one plan is cheaper in some regards, but uses more memory than the other. *CompareTotal*, when *having* to pick one of the two plans, can decide that since the memory is available, the first plan is chosen — or vice versa. If no such decision can be made, a heuristic can be employed.

Constant	Value
PAGE_SIZE	8 192 bytes
PAGE_READ	1.7 ms
PAGE_WRITE	1.8 ms
SEEK	9.5 ms
CPU_FACTOR	0.00005 ms

Table 4.1: Cost model constants

4.4.2 Rule Interface

Above, we presented the cost component interface for the plan generator. However, the rules representing the different physical operators also need some kind of interface to the cost component to be able to model the costs by inputting the costs of the operators constructed. This interface is not as formal as the former, since the cost model and rules are much more interdependent. While the optimizer only cares about comparing two plans, the rules must provide a reasonable model for the costs of its operators, and the cost model must support the modeling needs of the rules.

It does so by providing a set of cost primitives (as seen in Listing 4.2) the rules use to express the costs associated with applying the physical operator(s) expressed by the rules. Costs are stored as floating point number, a linear combination of CPU, I/O, communication costs and so on, which approximates the cost as time in 1/10 milliseconds. The different primitives like *ReadSequential*, *WriteSequential* all return the costs of the operation in question as a scaled floating point number which is this linear combination. As such, the implementation of *CpuCosts* is responsible for scaling the value returned by multiplying with a low factor, since executing a few CPU instructions is far, far faster/cheaper than reading from disk. By encapsulating the cost factors in each such primitive, it is easy to make changes without affecting the rules itself. For instance, switching from conventional disk drives to solid state drives would make random I/O-operations significantly faster. All that would be required is to alter the cost factor used inside the *ReadRandom* and *WriteRandom* primitives — which would be run-time configurable and not hard-coded.

The constants we use in our implementation is listed in Table 4.1 and are based on specifications for average hardware components. Also note how they are used in Listing 4.2.

For instance, a rule can express that the operator is expected to read X pages sequentially and execute Y CPU instructions for all n tuples. It can then get the costs for each such operation, add them to the costs of the input plan and annotate the resulting plan with the sum of the costs.

The *InputCosts* function allows a rule to express its costs as a function of how many times it reads its inputs. For instance, a nested loops join will probably read its left input once and its right input as many times as there are records in the left input. The left and right input plans and how many times they are read are passed to the function, and the costs are returned. As we will see later, this is not as simple as $left.Cost \times leftReads + right.Cost \times rightReads$ due to how costs propagate in DAGs.

4.5 Cost Models for MARS' Operators

We now continue by presenting how we have modeled the costs and effect on plan set state for the MARS operators we support. As previously mentioned, these are not particularly accurate, but good enough to demonstrate the concepts of the optimizer and produce decent plans.

```
1  public class BasicCost {
2     ...
3     public static double PageAccesses(double bytes) { Ceiling(bytes / PAGE_SIZE) };
4     public static double ReadSequential(double pages) { SEEK + pages * READ_PAGE };
5     public static double WriteSequential(double pages) { SEEK + pages * WRITE_PAGE };
6     public static double ReadRandom(double pages) { pages * (SEEK + READ_PAGE) };
7     public static double WriteRandom(double pages) { pages * (SEEK + WRITE_PAGE) };
8     public static double CpuCosts(double cardinality, double instructions)
9        { cardinality * instructions * CPU_FACTOR };
10    public static BasicCost InputCosts(Plan left, int leftReads, Plan right,
11                                       int rightReads)
12 }
```

Listing 4.2: Rule's cost model interface, simplified

We have not implemented a memory manager that distributes and sets memory limits for operators, so the MAX_MEMORY variables referred to below are currently static.

Lookup

cardinality, tupleSize = *from index stats*
cost = ReadSequential(PageAccesses(ResultSetSize))

Internally, Lookup uses B-tree lookups, but since we do not have access to its implementation, we have assumed a very simple cost model where data is read sequentially.

Sort

CPU_INSTR = 30
cardinality, tupleSize = *same as input*
cost = CpuCosts(Cardinality $\times$ $\log_2$(Cardinality), CPU_INSTR)
if ResultSetSize > MAX_MEMORY **then**
  cost += WriteSequential(PageAccesses(ResultSetSize))
  cost += ReadSequential(PageAccesses(ResultSetSize))
end if

The cost of the sort operator is the CPU cost of sorting, $\mathcal{O}(n \log n)$ (times a CPU constant). If the result set is larger than the maximum memory allocated to sorting, we assume it is enough to spill to disk once, and we add cost for writing and reading all the data once.

Selection

PRED_COST = 5
cardinality = input.Cardinality $\times$ Selectivity
tupleSize = input.TupleSize
cost = CpuCosts(cardinality, |Predicates| $\times$ PRED_COST);

We simply assume a static predicate cost for any kind of predicate and multiply it by the cardinality.

Group

STREAM_INSTR = 20, HASH_INSTR = 50, FIELD_WIDTH = 20

```
cardinality = input.Cardinality × Selectivity
tupleSize = (|GroupFields| + |Aggregators|) × FIELD_WIDTH
if input's grouping satisfies GroupFields then
   cost = CpuCosts(input.Cardinality, (|GroupFields| + |Aggregators|) × STREAM_INSTR);
else
   cost = CpuCosts(input.Cardinality, (|GroupFields| + |Aggregators|) × HASH_INSTR);

   if ResultSetSize > MAX_MEMORY then
      cost += WriteRandom(PageAccesses(input.ResultSetSize))
      cost += ReadSequential(PageAccesses(input.ResultSetSize))
      cost += CpuCosts(input.Cardinality, (|GroupFields| + |Aggregators|) × HASH_INSTR));
   end if
end if
```

The costs for the group operator depend on whether the input stream is already grouped by the required attributes. The rule asks the orderings- and groupings component if it is and calculates cost accordingly. If it is grouped already, we only need to stream the tuples through and accumulate values in the aggregates, and we use a low constant for CPU cycles per tuple. If not, we need to hash the tuples, using a higher CPU constant. We may also need to spill to disk if the result set is too large. For output tuple size, we simply assume a static field width.

Map

```
EXPR_WIDTH = 20, EXPR_COST = 5
cardinality = input.cardinality
tupleSize = input.TupleSize + |Expressions| × EXPR_WIDTH
cost = CpuCosts(cardinality, |Expressions| × EXPR_COST);
```

Since the map operator is one which can call user defined functions, the cost of those must be handled in some way. Today, we simply assume a static predicate cost for any kind of expression. For output tuple size, we simply assume a static field width.

HybridHashJoin

```
HASH_INSTR = 35
cardinality = left.Cardinality × right.Cardinality × Selectivity
tupleSize = left.TupleSize + right.TupleSize
cost = CpuCosts(cardinality, HASH_INSTR)
if left.ResultSetSize > MAX_MEMORY ∧ right.ResultSetSize > MAX_MEMORY then
   cost += WriteRandom(PageAccesses(left.ResultSetSize + right.ResultSetSize))
   cost += ReadSequential(PageAccesses(left.ResultSetSize + right.ResultSetSize))
   cost += CpuCosts(left.Cardinality + right.Cardinality, HASH_INSTR);
end if
```

For a hybrid hash-join, the base cost is the output cardinality times a constant. If both inputs are larger than the available memory, we need to spill to disk, and this adds additional write, read and CPU costs for rehashing the data.

Cost for MergeJoin

```
MERGE_INSTR = 20
```

```
1 public struct BasicCost : ICost {
2    public BasicCost(double firstRead, double furtherReads) { ... }
3    ...
4    public double FirstRead { get; set; }
5    public double FurtherReads { get; set; }
6    public int Passes { get; set; }
7 }
```

Listing 4.3: Cost model state.

cardinality = left.Cardinality × right.Cardinality × Selectivity
tupleSize = left.TupleSize + right.TupleSize
cost = CpuCosts(cardinality, MERGE_INSTR);

For a merge-join, the cost is the output cardinality times a smaller constant than for hash join. This is not very accurate, however. Since MARS supports operators propagating skipping to its input nodes, the true cost is a lot less when merging a small and a large input where large parts of the large input is skipped.

ScoreOccurrences

CPU_INSTR = 30
cardinality = Σ children.Cardinality
tupleSize = firstChild.tupleSize
cost = CpuCosts(cardinality, CPU_INSTR);

The cost is simply CPU costs proportional to the cardinality.

4.6 Cost Model Implementation

So far, we have seen how the plan generator and rules use the cost model. We now shift the focus to how it is implemented internally, and how the costs for a plan is represented.

4.6.1 Cost State

To be able to reason about costs, the optimizer needs a basic unit of cost. The plan generator does not care about what this unit is, only that it implements the ICost interface as explained in Section 4.4.1. Whenever a plan is created, it is annotated with its cost using the ICost implementation.

In our simple cost model implementation, this basic unit of cost is the struct BasicCost as shown in Listing 4.3.

The two data members $FirstRead$ and $FurtherReads$ store the plan costs as the aforementioned linear combination of the different cost categories. The reason for using *two* numbers, and not one, is that sometimes it is cheaper to execute a plan (read its output) more times when already done once. A good example is an operator that materializes its output, such as a hash group. Multiple reads typically happen when using nested loop joins — although not currently supported by MARS, the cost model supports it. $FirstRead$ is the cost of the first read, while $FurtherReads$ is the cost for each additional read. $Passes$ is used when calculating costs for DAGs — this is described later in this section. $FurtherReads$ is present to facilitate future support for more fine-grained cost modelling, but currently all rules simply set this to $FirstRead$, since the implemented operator cost models are quite simple.

```
public CostRelation Compare(ICost other) {
   if (FirstRead < other.FirstRead && FurtherReads <= other.FurtherReads)
      return CostRelation.Better;
   if (FirstRead > other.FirstRead && FurtherReads >= other.FurtherReads)
      return CostRelation.Worse;
   if (FirstRead == other.FirstRead) {
      if (FurtherReads < other.FurtherReads)
         return CostRelation.Better;
      if (FurtherReads > otherCost.FurtherReads)
         return CostRelation.Worse;
      return CostRelation.Equal;
   }
   return CostRelation.Unknown;
}

public CostRelation CompareTotal(ICost other) {
   CostRelation result = Compare(other);
   if (result == CostRelation.Unknown) {
      if (FirstRead < ((BasicCost)other).FirstRead)
         return CostRelation.Better;
      else
         return CostRelation.Worse;
   }
   return result;
}
```

Listing 4.4: BasicCost's ICost implementation.

If we were to implement a more detailed cost model, we would probably go for a vector model in which the different classes of cost (CPU, IO, memory network, etc.) are separated. However, this complicates the decision of which plan is cheapest (a plan could be cheaper in I/O, but use more CPU), so we chose to stick with our simpler model.

4.6.2 Cost Comparison

Listing 4.4 shows BasicCost's implementation of the ICost members, namely *Compare* and *CompareTotal* as previously described.

The implementation of *Compare* is straightforward. If both $FirstRead$ and $FurtherReads$ are lower or equal, the plan is *better* (except if both are equal where the plans are equal), otherwise it is *worse*. The only exception is if $FirstRead$ is lower and $FurtherRead$ higher or vice versa — then the result is *unknown* since none of the plans are "cheaper".

CompareTotal utilizes the implementation of *Compare* and checks for the special case of Unknown. In that case, it uses a simple heuristic: It is assumed that the plan will only be executed once and then compares $FirstRead$ for the two plans.

4.6.3 Cost Calculations

During plan generation, rules combine operator costs with costs of sub-plans to form a new BasicCost instance. To make this neat, we have overloaded the $+$ and $*$ operators for BasicCost as shown in Listing 4.5. Summarizing costs is straightforward; $FirstRead$ and $FurtherReads$ are summarized independently. The $*$ operator is used when reading a plan multiple times, i.e. $newCosts = A.costs * n$, and is more complex since it must take into account that the first read might be more expensive than further reads. The resulting value for $FirstRead$ is therefore the cost of reading the input plan once, plus the number of additional times (which

```
1  public static BasicCost operator +(BasicCost a, BasicCost b) {
2     return new BasicCost(a.FirstRead + b.FirstRead,
3                          a.FurtherReads + b.FurtherReads);
4  }
5  public static BasicCost operator *(BasicCost input, double times) {
6     return new BasicCost(input.FirstRead + (times - 1) * input.FurtherReads,
7                          times * input.FurtherReads);
8  }
```

Listing 4.5: Overloaded operators for BasicCost

```
1  public class AbstractRule {
2     public virtual void UpdatePlan(Plan plan) {
3        BasicCost cost = Cost(plan);
4        if (plan.Children.Count == 1)
5           cost += plan.OnlyChild.Costs;
6        else if (plan.Children.Count > 1)
7           cost += BasicCost.InputCosts(plan.Children, null);
8        plan.Costs = cost;
9     }
10
11    protected abstract BasicCost Cost(Plan plan);
12    ...
13 }
```

Listing 4.6: Default implementation of *UpdatePlan*

is $times - 1$) it is to be read, multiplied by the input plan's $FurtherReads$. The resulting value for $FurtherReads$ is just $times$ multiplied by the input plan's $FurtherReads$, since we have already read the input plan at least once already.

Whenever a new plan is constructed by a rule, its costs need to be set. This happens in the *UpdatePlan* method, which the abstract base class AbstractRule provides a default implementation of. It also defines an abstract function *Cost* which the rules needs to implement to return the cost of the operator they represent. If the rule has no sub-plans, the resulting cost is only the cost of the operator itself. If it has one sub-plan, the cost is the sum of the costs of the sub-plan and the operator. If the rule has more than one sub-plan, we have the possibility of DAGs and we need to use the *InputCosts* function as described in the next section.

4.6.4 Cost Calculation for Trees vs DAGs

Calculating the cost for the resulting plan when adding a unary operator (one input) on top of an existing plan is simple enough: Take the costs of the input plan (the sub plan with the new operator now being added on the top), add (sum) the costs of the new operator, and get

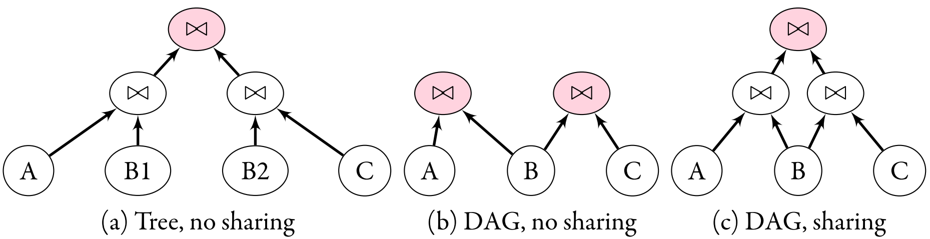


Figure 4.1: Three cases, with the nodes being costed are red

```
1 public static BasicCost InputCosts(Plan left, int leftReads, Plan right, int rightReads) {
2    if (!left.Sharing.Overlaps(right.Sharing))
3       return left.Costs * leftReads + right.Costs * rightReads;
4    else
5       return InputCostsDag(left, leftReads, right, rightReads);
6 }
```

Listing 4.7: *InputCosts* function

the resulting cost of the new plan. For rules with more than one input, it also seems simple enough: Summarize the costs of the input plans and add the costs of the new operator. This approach works well if the input plans are disjoint (the resulting plan forms a tree) as shown in Figure 4.1a, but if the resulting plan forms a DAG with a shared sub-graph, as in Figure 4.1c, it will not work.

This is due to the fact that the cost of the shared sub-plan (B in the figure) will be counted twice in this situation, while it should only be counted once, since this plan is only executed once even if it is read twice. One might think that if sharing is present, this should already have been reflected by having lower costs on the sub-plan, but the trick to understand this is that *the sharing appears now*. Even if two plans share some common sub-plan, they are just two loose plans (Figure 4.1b). The *actual sharing*, i.e. the reuse of the sub-plan output, does not occur before these two plans together form a larger plan. This can also happen for multi-queries, as a multi-query can be viewed as one large query with the output operator on top.

This means that whenever a plan is formed that has more than one sub-plan, we must be careful about how we calculate its costs. If all the sub-plans are completely disjoint, i.e. we are now forming a tree, we can calculate the costs normally, as no sharing is possible. If, however, some of the sub-plans overlap, i.e. we are now forming a DAG, it means that sub-plans are shared and we need to employ a different strategy for cost calculation to recognize sharing and reduce plan costs accordingly.

This brings us to the *InputCosts* function in Listing 4.7, which handles the logic described above. The parameters are the left input plan, how many times it is read/executed, and the same for the right plan. We have simplified its signature to make it easier for the reader to follow, so it does not match the signature shown in Listing 4.6 completely, but the point remains the same. It examines the `Sharing`-properties of both plans to detect if they overlap. If they do not, costs are calculated the obvious way, utilizing the overload of the $*$ operator as previously described. If they *do* overlap, however, it will delegate the work to the *InputCostsDag* function, which is described in the next section.

Cost Calculation for DAGs

We now move on to the missing piece; how costs are calculated for DAGs. Whenever costs are calculated for an n-input operator (with $n > 1$) with overlap between its inputs, a recursive traversal is performed of the plan graph top-down, starting at the operator being calculated costs for. This traversal is done by the rules themselves in the *DagCosts* function they all need to implement. Its signature is shown below. It returns the cost of reading/executing $plan$ – $reads$ number of times. In most cases, $reads$ will be 1, except if we are dealing with nested loops joins.

```
ICost DagCosts(Plan plan, int reads);
```

Let us use Figure 4.1c as a starting point. When calculating the costs for the topmost operator, *DagCosts* will be called recursively down the left and right input. The key here is that *DagCosts* will be called twice for the B plan, once from the left output and once from the right.

```
private static BasicCost InputCostsDag(Plan left, int leftReads, Plan right, int rightReads) {
   BasicCost result = left.Rule.DagCosts(left, leftReads) + right.rule.DagCosts(right,
       rightReads);
   resetPasses(left);
   resetPasses(right);
   return result;
}

private static void resetPasses(Plan plan) {
   if (plan.Costs.Passes != 0) {
      plan.Costs.Passes = 0;
      foreach (Plan subPlan in plan.Children)
         resetPasses(subPlan);
   }
}
```

Listing 4.8: Implementation of *InputCostsDag*

It is here the Passes variable in the cost state for a plan (shown in Listing 4.3) is necessary — it is used by *DagCosts* to keep track of the number of times this plan has been executed already. The first call will calculate and return costs normally and at the same time set the $Passes$ variable. The next time it is called, it can check $Passes$ to see if additional executions are required. If it is not, it is essentially (almost) free to read the results a second time and the cost returned to the right output would be zero. Of course, the true cost is that of the Copy-operator, but since it is cheaper than re-executing anyway, we need not dwell on better estimation.

Note that this requires the $Passes$ variable to be reset after each cost calculation to not affect future calculations for operators closer to the root in the same query. This is handled by the *InputCostsDag* function, which first starts the recursive traversal for the left and right plan. Afterwards, it calls *resetPasses* to reset the $Passes$ variable. Note that $Passes$ is initially set to 0. *resetPasses* contains a little optimization: No plan with $Passes = 0$ can have a plan with $Passes \neq 0$ below it.

Let us now look at some actual implementations of *DagCosts*. We start out with the default implementation from AbstractRule in Listing 4.9 which works for all operators that do not spool their results to disk. The Cost abstract function returns the costs of executing the current operator once, as before.

First, *DagCosts* checks if $reads$ is larger than $Passes$. If it is not, it means that we have already executed this plan enough times. This happens if one of the other operators receiving data from this plan have requested $Passes$ executions. In that case, it is cheap to read the results $reads$ times, so *zero costs* is returned. If, however $reads$ is larger than $Passes$ (this could happen if this is the first call and $Passes$ is still 0, for instance), we calculate the $additional$ number of passes required. If not dealing with nested loops join, the most common case is $Passes = 0, reads = 1$, so $additional = 1$. $Passes$ is then set to $reads$ for later calls to *DagCosts*. Then, costs for executing this operator $additional$ number of times is calculated, followed by adding the costs of executing each of the input plans $reads$ times (this is the recursive call). Finally, the result is returned.

For operators spooling their output to disk, it is implemented a bit differently. It will only calculate its own costs, and the costs of its sub-plan for the first call to *DagCosts* ($Passes = 0$). For all subsequent calls, only the costs to read the already produced results from disk is considered.

This approach to DAG-costing performs well in most cases, but can have exponential runtime in the special case of a nested loops join chain. However, we will not hit this case with the

```
1  public class AbstractRule {
2     protected abstract BasicCost Cost(Plan plan);
3     ...
4     public virtual ICost DagCosts(Plan plan, int reads) {
5        if (reads > plan.Costs.Passes) {
6           int additional = reads - plan.Costs.Passes;
7           plan.Costs.Passes = reads;
8
9           BasicCost result = Cost(plan) * additional;
10          foreach (Plan subPlan in plan.Children)
11             result += subPlan.Rule.DagCosts(subPlan, reads);
12          return result;
13       }
14       return BasicCost.Zero;
15    }
16 }
```

Listing 4.9: Default implementation of *DagCosts*

current set of operators supported by MARS, so we have chosen this approach due to its simplicity. For a more detailed discussion of this and other approaches to DAG costing, see [Neu05].

4.7 Example Cost Calculation

To illustrate how the cost calculations work, we will now perform the cost calculations for two versions of a simple query; one without (Figure 4.2a) and one with sharing (Figure 4.2b). Note that formula parts in upper case are constants.

We start with the one without sharing. First, the base plans (square corners) are created, costed and entered into the memoization table. Their cost depend only on the index statistics fetched from the system catalog. Since there is no buffering in MARS, first and further reads cost the same. Lookups with large result sets are more expensive, and the two equivalent lookups on *olstad* cost the same. By using Table 4.1, Listing 4.2 and the cost model for Lookup previously given, we find that the cost for a lookup is

$$\text{SEEK} + \left\lceil \frac{Cardinality \times TupleSize}{PageSize} \right\rceil \times \text{READ_SEQUENTIAL}$$

so the cost for the lookups on *olstad* becomes

$$95 + \left\lceil \frac{500 \times 100}{8192} \right\rceil \times 17 = \underline{214}$$

The calculations for the other lookups are done in the same way. Then the plan with the left join on top is constructed. Since there is no overlap between the input plans, its costs are simply the sum of the costs of the two lookups, plus the cost of the join itself:

$$LeftInput + RightInput + (OutputCard \times \text{MERGE_INSTRUCTIONS} \times \text{CPU_FACTOR})$$

$$= 214 + 690 + (500 \times 20 \times 0.0005) = \underline{909}$$

Merge joins are quite cheap, so the costs of the lookups (I/O) dominate. First and further reads have the same cost. The tuple size is the sum of the input tuple sizes. We assume that there exists a foreign key relationship from *Occurrence1* (left lookup) to *Occurrence2* (right lookup)

although MARS has no such information, so the cardinality is the same as the left input. The right join plan is constructed the same way.

This is a multi-query, and the cost of the entire query is the cost of the output operator. Since its two input plans do not overlap, its cost is simply the sum of the costs of the input plans, i.e. $909 + 484 = \underline{1393}$.

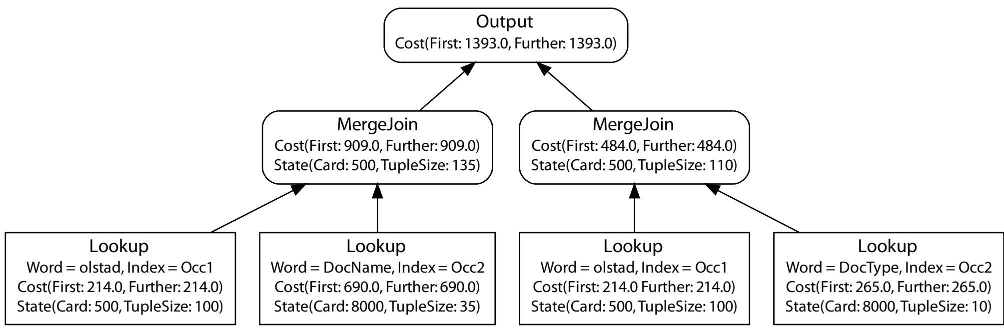
We now move on to the query with sharing and highlight the differences. The base plans are constructed and costed in the same way. When we move on to the joins, everything is the same as for the tree case. Even though the lookup in *olstad* is shared between the input plans, the join plans are separate plans and the left and right inputs to each of the joins do not overlap. Hence, the costs of the input plans are just summarized and we get the same cost for the join plans as in the tree case.

However, when we move on to the costing of the output operator, things change. When the rule for the output operator invokes *InputCosts* to find the *globally* cheapest plan (which in many cases is *not* is the combination of the cheapest of each sub plan), *InputCosts* will now detect that the left plan and right plan overlap (i.e. *olstad* is shared between them) — and conclude that we have a DAG query with sharing. Instead of just summarizing the costs, it will instead invoke *DagCosts* for the left and right input plan (the left and right join) — i.e. the cost of the entire query now becomes $DagCosts(leftJoin, 1) + DagCosts(rightJoin, 1)$.

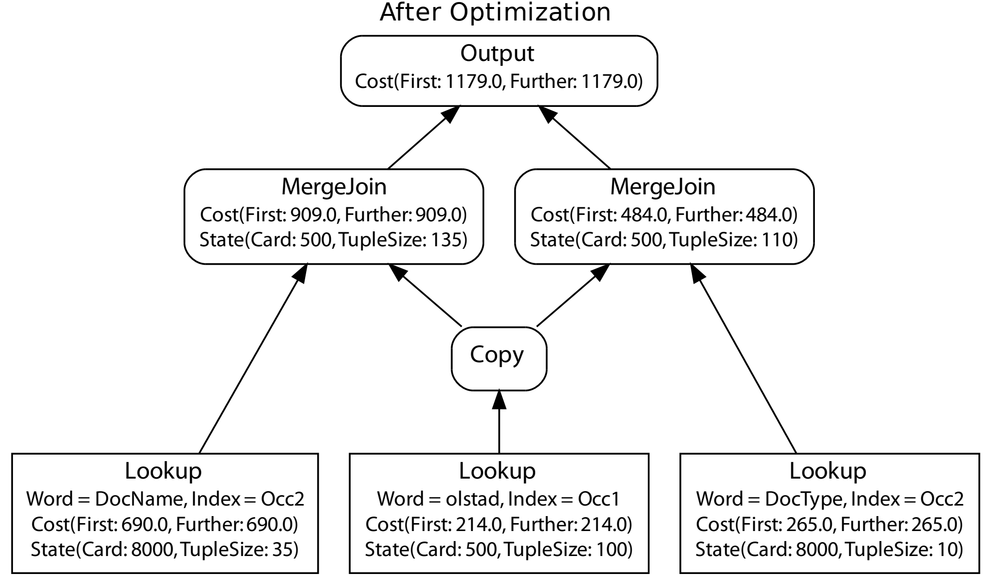
DagCosts on the left join will in turn call *DagCosts* on the two lookups below it, which will return the same costs as can be seen in the figure. *DagCosts* on the left join will sum these two and add its own costs, and return the cost seen in the figure. But, at the same time the $Passes$ **variable have been set to** 1 for the left join and the two lookups below it.

Next, *DagCosts* is called for the right join, which in turn calls *DagCosts* on the shared lookup. *DagCosts* for this lookup will now realize (by examining $Passes$) that no further work is required and **return *zero costs***. This is summed with the costs of the right lookup and the join itself and returned. Thereby, the costs of the shared lookup is only counted once, not twice. We assume that the copy operator is free, although this is not completely true. However, even if we added its cost, it would still be cheaper than performing the lookup twice — and it is the relative numbers that matter, not the absolute. This may not be true if the copy operator has to spool to disk if the output consumes data at a very different pace, but we have chosen to look away from this problem for now. The final cost of the query is therefore Left sub tree + Document type lookup + Right join $= 909 + 265 + 5 = \underline{1179}$. As expected, this is equal to the cost of the tree query minus the cost of the shared lookup (since it is only executed once): $1393 - 214 = \underline{1179}$.

By using the algorithms and techniques described in this chapter, the plan generator can get an estimate for the cost of the plans it generates. We know move on to what drives the search and creates the plans to be calculated costs for – namely the rules.



(a) Tree, no sharing



(b) DAG, sharing

Figure 4.2: Example cost calculation for tree vs DAG for the same query

5 Rules: Search Space and Pre-/Post Processing

"Any problem in computer science can be solved with another level of indirection."

– David Wheeler

5.1 Introduction

As introduced in Section 2.4, rule-based optimization is much more extensible than static, hard-coded optimizers. This chapter introduces the rules used in our optimizer and equally important: how they are integrated with the optimizer.

Rules provide extensibility and modifiability in the sense that the optimizer does not know the individual rules specifically. It only knows the rule population and applies the rules applicable at any given moment. To be able to do this, the optimizer must have a common interface to all rules. For example, all transformation rules implement ITransformationRule, whereas rules used during the search phase implement either the IBaseRule- or ISearchRule interfaces.

Another important matter is how the optimizer is made aware of the rules. It clearly cannot be hard-coded in the optimizer itself, since this would undermine extensibility. This means that the optimizer cannot know the rules at compile time. Instead, the optimizer uses *reflection*, which is a feature in .NET for reasoning about program metadata. At optimizer start up, using reflection, all types (classes) in all known assemblies (DLLs, .NET equivalent of Java JARs) are enumerated. Those identified as rules are loaded into the optimizer and prepared for optimizer use. Reflection used like this is somewhat expensive, but since this only happens once at system start-up, it is not a problem. See Section B.2 for the implementation of this step.

To be taken into consideration for optimizing, all the rules have to do is to declare themselves as rules. To do this, we use custom .NET attributes, which is a way to annotate classes (and any other programming construct) with metadata. For transformation rules, this happens by appending e.g. [Preprocessor] before the class declaration. Constructive rules use a rule binder concept as explained in Section 5.3.1. Thereby, the optimizer does not have dependencies on the rules, and minimal effort is needed to add new rules — it just needs to be tagged, compiled and made available to the optimizer.

We first present the transformation rules used during pre- and post-processing, then the constructive rules used during plan generation. Whereas transformation rules transform a complete operator graph from one valid state to another, constructive rules build the graph from scratch.

5.2 Transformation Rules

This section briefly explains how some pre- and post-processors are implemented, since some vital steps of the optimization are implemented as -processors. They are transformation rules that comply to the interfaces and semantics described in Section 3.5.

5.2.1 Pre-Processors

Annotators

First, we describe some "Annotators". Annotators do not (by convention) change the graph structure. They are intended to incrementally supply the nodes with more information. Instead of doing everything in one pass of the graph, we split them into separate components, that are more readable and maintainable.

Behaviour-, Produces-, Dependency- and EquivalenceAnnotator are four pre-processors that walk the query graph in topological ordering, dispatching calls to rule-binders that have declared being a mapper of the visited node. This is because the optimizer is oblivious to the semantics of the operators. We want any operator specific behaviour to be defined in their corresponding rules and/or rule binders.

BehaviourAnnotator sets the operator's "behaviour" — that is whether the operator changes the record structure, the amount of tuples, its ordering/grouping, etc. For example, a selection operator changes the tuple amount, but not the ordering. A map operator can change the record structure, but not the amount or ordering. A group operator potentially changes both tuple amount, ordering, grouping and record structure.

This information is necessary because some transformations need to reason about the behaviour. For example, joins that only changes the record structure and not the result *set* can be reordered more freely.

ProducesAnnotator maps the origins of attributes used in operators. The purpose is to always know where an attribute originates, to be able to correctly map operator dependencies. Also, attributes in expressions (such as in Map and Select) are altered to unambiguously refer to the attribute of a certain node using NodeAttribute instances instead of names. A NodeAttribute is basically a (Node, Attribute) tuple.

Problems that can occur if we do not deal with this rigorously are not only name clashes, but also wrongly named attributes, as some operators in MARS (specifically joins) alter the attribute names in different ways depending on their order.

DependencyAnnotator sets the operator's dependencies. Operators trivially depend on the other operators that produce the attributes they require. However, dependencies are also used to force placement of e.g. outer joins, trims, and so on — as explained in Section 3.4. DependencyAnnotator depends on ProducesAnnotator being run.

EquivalenceAnnotator marks which attributes are deemed equivalent. For example, an equijoin of two attributes will induce that the two are equivalent. The equivalence classes this annotator returns are needed to be able to split up join pairs and freely reorder them. We describe this further in Section 8.5.5.

CopyRemover

To keep rules and processors simple, we convert input query DAGs into trees. Common subexpressions are recognized during the search phase regardlessly, so no optimization opportunities are lost.

Thus, there is no Copy-operator or -rule-binder. See Section 5.2.2 for the opposite — a post-processor that inserts Copy-operators for operators with multiple outputs.

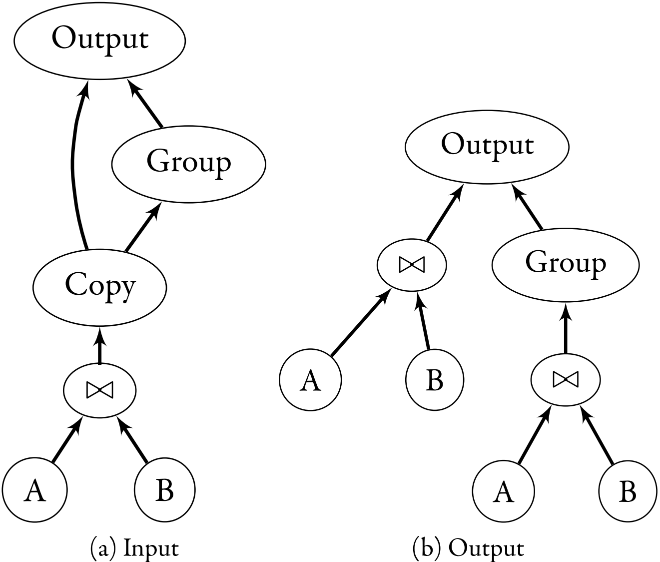


Figure 5.1: Query Graph Before and After CopyRemover

Figure 5.1 illustrates how this works.

OrderingExtractor and SortRemover

Our optimizer treats orderings as a *desired property* of the input stream to an operator — and *not* as an operator. This is explained in depth in Chapter 6.

OrderingExtractor analyzes whether the output of a query (or queries, in a multi-query) should have a specific ordering — i.e. that the ordering of the output of a Sort-operator is *guaranteed* to be preserved until it reaches the Output-operator. This is the case for e.g. a query with a Sort-operator with only "order preserving" operators between the Sort and Output. For example, a Select over a Sort does not alter the order of the tuples. However, a logical Join over a Sort might, as a HybridHashJoin might be chosen.

When an ordering for a result set is desired, the relevant Output-inputs (i.e. queries) are marked as such. In addition to the Output operator, Trim also needs to have any guaranteed input orderings preserved.

SortRemover iterates over all SortOperators after OrderingExtractor has run — either removing the operator or replacing it with a TrimOperator, whenever the SortOperator also trims the output.

LogicalJoinTransformer

LogicalJoinTransformer iterates over all HybridHashJoin- and MergeJoin-operators, replacing them with a logical Join-operator. MARS does not have a logical join operator, and it is possible that the join-implementation specified in the input query is not the optimal one.

The full implementation can be found in Section B.8.

MultiJoinSplitter

The MergeJoinOperator in MARS supports joining more than two input relations. To keep rules and patterns simple, the MultiJoinSplitter-pre-processor splits an n-way MergeJoin into $n - 1$ binary left-deep logical joins.

This pre-processor was written before we decided on the approach for equivalence class joins described in Section 8.5.5. It is currently disabled due to reasons explained there. We have not studied that particular problem thoroughly enough to decide whether keeping the

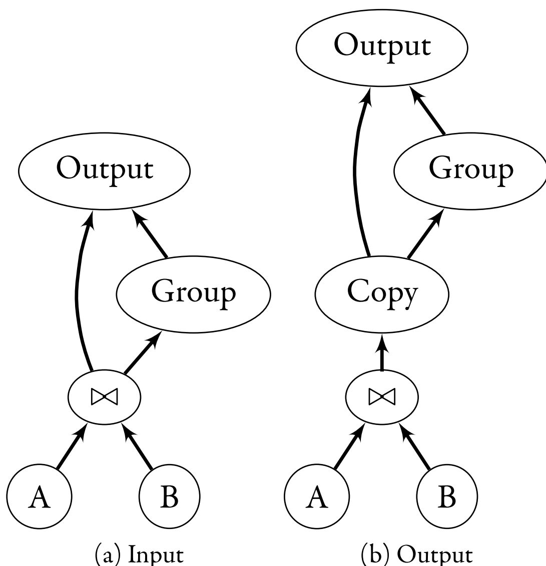


Figure 5.2: Query Graph Before and After CopyInserter

processor is a good approach — it could possibly make rules and patterns simpler. Currently, we only optimize binary joins, but consider support for n-way merge joins throughout the report, to avoid decisions that prevent future support for it.

5.2.2 Post-Processors

CopyInserter

MARS' Copy-operator is the only operator that can have multiple outputs.

CopyInserter is a post-processor that inserts Copy-operators over operators with multiple outputs. The Copy-operator is set as the only output, and the inputs of the parent nodes are replaced with input from the inserted Copy-operator. Figure 5.2 shows an example transformation.

FieldMapper

To avoid problems with attribute name clashes, all references to attributes are tuples with a reference to the node producing the attribute, and the name of the attribute. However, before returning to MARS, we need to set the names of the attributes the various operators produce. This is done by dispatching *SetFields()*-calls to the rule binders. The rule binders then generate a unique attribute name and map their (node, attribute)-tuples to the unique attribute names.

Finally, to ensure that the user gets the same attribute names and attribute order as in the input query, Map-operators are added that rename the unique aliases to the expected names in the correct order. This is necessary not only because of name clashes that could occur when the graph is altered, but also because certain MARS operators alter the attribute names. For example, the join operators prefix the non-key attributes with "Input0" and "Input1". Thus, a join-ordering that differs from the one in the input query will have different attribute names, which consequently need renaming.

Merge Trim and Sort

Observing that a sort operator with a trim just above it is semantically equivalent to setting the trim options on the sort operator and removing the trim operator, we have developed a rule to

rewrite this particular situation in the operator graph.

Actually, the trim operator does not even have to be just above the sort operator. It is semantically correct to merge them as long as none of the following types of operators are between them:

Operators altering the tuple set. Operators altering the tuple set by removing or adding tuples will change the tuples produced by the trim operator, causing behavior that cannot be predicted.

Operators altering the tuple order. Obviously, altering the order of the tuples will alter the output of the trim operator as well.

Double output operators. If the output from the sort operator is used by other branches in the graph, it cannot be modified.

A special case arises when the trim properties are already set on the sort operator. They need to be combined with the properties on the trim operator. We have omitted the details for brevity.

5.2.3 Additional Transformations

During our research, we have identified various transformations usually done by query optimizers like [Pos08b]. In the following, we describe some of them. Although we have not implemented them, and some are specific to SQL, we include them as inspiration for future work.

Replace plans that produce no output with a no-op. If a plan is guaranteed to produce no output (like a `SELECT TOP 0`), it can be replaced with a dummy operator that produces no output.

Evaluate constant expressions. This step involves evaluating any expressions that turns out to be constant — i.e. expressions that are only built up from constant sub expressions.

Transform ANY and EXISTS in WHERE and JOIN/ON clauses to joins, if possible.

Reducing outer and semi join to inner joins can be beneficial where possible. See [HR] for an example.

Constraint exclusion enables the optimizer to use constraints to optimize the query. This is particularly useful when relations are partitioned. For instance, there is no point in searching for events that happened in 2008 in a partition containing only events for 2007.

Except Conditions. Push conditions from the first operand of EXCEPT into the second operand as well (we will not need the extra results anyway). The same goes for INTERSECT.

Transform MIN/MAX aggregate functions. Sometimes it is beneficial to replace MIN/MAX aggregate functions by sub-queries of the form `SELECT col FROM tab WHERE ... ORDER BY col ASC/DESC LIMIT 1`.

Split selection predicates. Selection predicates in conjunctive normal form can be split to be able to move them separately around the operator graph. If not in CNF, they can possibly be transformed.

Push NOTs down as far as possible. Apply DeMorgan's laws if applicable.

Distinct push down vs. pull up vs. elimination. Push down: Allow early elimination of duplicates. Pull up/elimination: Due to implicit distinctiveness from joins, etc.

Transitive closure of predicates. For instance, given that we have `T1.C1 = T2.C2, T2.C2 = T3.C3, T1.C1 > 5`, we can also add `T1.C1 = T3.C3 AND T2.C2 > 5 AND T3.C3 > 5` to increase selectivity.

Merging sub-queries. In some cases, multiple sub-queries can be merged to a single sub-query. For example, if multiple sub-queries fetch data from the same table, a merge may be possible.

Inline functions. It may be beneficial to inline functions, i.e. make sub-queries of them.

For more transformations and rewrite rules and techniques, see [Moe06], part III.

5.3 Constructive Rules

Constructive rules are used during the plan generation step, and are each responsible for constructing a part of the query graph. They declare what part of the query they can be responsible for, enabling the plan generator to determine which rules to utilize.

It is important to realize that adding multiple alternative rules for the same query parts will increase the size of the search space accordingly. However, this is often necessary, for example when there are multiple access paths.

The rules decide themselves in which direction they want to take the plan search and how they would like to build the query graph. They are not transformative, and are not used during pre/post-processing. The rules come in different types, depending on their role in the plan generation.

We start out by formalizing the types of rules and their interface to the plan generator. Then, we explain the different rules we have implemented.

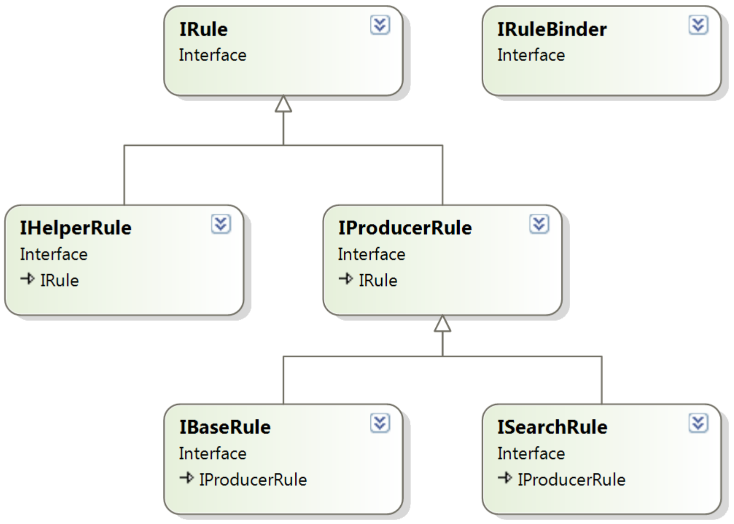


Figure 5.3: Class diagram for the constructive rule interface hierarchy.

```
1  public interface IRule {
2     void UpdatePlan(Plan plan);
3     OperatorNode BuildAlgebra(Plan plan);
4     ICost DagCosts(Plan plan, int reads);
5  }
6  public interface IProducerRule : IRule {
7     BitSetProxy Produced { get; set; }
8     BitSet Filter { get; }
9     int Id { get; set; }
10    IList<OperatorNode> Nodes { get; set; }
11    bool StructurallyIdentical(IProducerRule other);
12    void MapShareEquivalentAttributes(IProducerRule from, Dictionary<NodeAttribute, NodeAttribute
          > attributeMappings);
13 }
14 public interface IBaseRule : IProducerRule {
15    void Initialize(PlanSet plans);
16 }
17 public interface ISearchRule : IProducerRule {
18    bool IsRelevantTo(BitSet goal);
19    void Search(PlanSet planSet, ICost limit);
20    IList<BitSetProxy> Required { get; }
21 }
22 public interface IHelperRule : IRule {
23 }
24 public interface IRuleBinder {
25    AbstractNodeMatcher Pattern { get; }
26    IEnumerable<IProducerRule> GetRules(QueryOptimization queryOptimization, IEnumerable<
          OperatorNode> matches);
27 }
```

Listing 5.1: Rule interfaces

5.3.1 Rule Interfaces and Rule Binders

The borderline between the plan generator and the different search rules includes multiple interfaces in a hierarchy, as shown in Figure 5.3. Their definitions are shown in Listing 5.1. A typical constructive rule will implement either IHelperRule, IBaseRule or ISearchRule, depending on the type of rule. IRule is a base interface for the rest, while IProducerRule is a base interface for all rules producing bit properties. *Rule binders* are used to instantiate rules. Below follows a description of each interface.

IRule, is the base interface for all constructive rules. The *UpdatePlan* method is used during plan generation and updates a given plan's cost, ordering, sharing and rule instance. *BuildAlgebra* is used during the reconstruction phase when each rule recursively constructs the final operator graph. *DagCosts* is invoked when the cost model has figured the plan is a DAG, as explained in Section 4.6.4.

Helper rules, which implement IHelperRule, are rules the optimizer does not directly know. Helper rules are typically used by other search rules. For instance, the Join rule uses the HybridHashJoinRule and the MergeJoinRule. Helper rules do not have any members in addition to IRule, so the interface declaration is empty and is just used as a marker.

Producer rules are used actively by the query optimizer during the search phase of the plan generation. As the name suggests, they produce bit properties, but do not necessarily require any properties. The Produced property gives which properties this rule can be used to achieve. `Filter` is used by the query optimizer to filter out inapplicable rules and unreachable plans, as described in Section 3.6.5. Each producer rule is given an `Id` to identify it to the

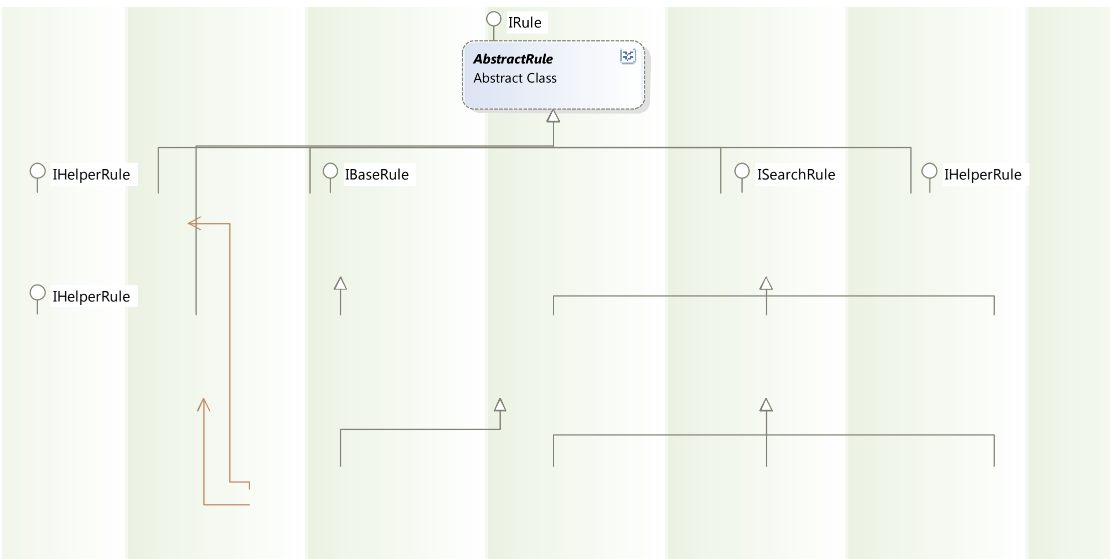


Figure 5.4: Class diagram for the implemented constructive rules

optimizer, and the `Nodes` property points to the original nodes in the input operator graph it was instantiated from. The two methods *StructurallyIdentical* and *MapShareEquivalentAttributes* are used to determine share equivalence, as explained in Section 3.7.

Base rules are rules producing, but not requiring bit properties. They are used to model the leaves of the operator graph, typically table-, index- or file scans. An example is the Lookup rule. The *Initialize* method is called when the rule is initialized during the preparation phase of plan generation.

Search rules can both produce and require bit properties. They model the internal nodes in the operator graph. Examples include Selection and Join. More importantly, they control the direction in which the optimizer searches for plans. The *IsRelevantTo* method determines if the rule instance is relevant to the goal specified, that is, if the rule can be useful in this context. The *Search* method is the heart of search rules. Whenever the optimizer finds that the rule instance is applicable in a certain context, it will call *Search* with a PlanSet. The rule should answer by generating plans satisfying the properties of the PlanSet (a BitSet). Normally, this will be done by applying some logic and calling back to the optimizer. A cost limit is included, and the search should be aborted whenever this limit is exceeded. Finally, the `Required` property returns a list of the required input properties for each child, from left to right.

An overview of the implemented constructive rules is shown in Figure 5.4. AbstactRule, AbstractBaseRule, AbstractSearchRule, UnaryRule and BinaryRule are abstract rules implementing common functionality. We spend the rest of this chapter explaining the details for the different rules. In the following, we have removed access modifiers, property getters and setters and most method implementations for brevity. We only list the most important class members. We have also left out the `Produced` property for base rules and the `Produced` and `Required` properties for search rules, as this is something all constructive rules implement.

Finally, **rule binders** provide the optimizer with a way to instantiate relevant rules for the query to be optimized. Classes that implement IRuleBinder, and are tagged with the [RuleBinder] attribute will be loaded by the optimizer upon start up.

The basic functionality of a rule binder is to declare a `Pattern` that matches nodes in the input operator graph and instantiate and return rules. The optimizer invokes the rule binder with a reference to itself and with a list of all the matches of the pattern. It is the entire responsibility of the rule binders to correctly configure the rules so they correspond to the operators in the query graph.

```
1 public class LookupRule : AbstractBaseRule {
2    BasicPlanSetState Stats;
3    string Index;
4    string Word;
5    OrderProxy Order;
6    void Initialize(PlanSet plans);
7    void UpdatePlan(Plan plan);
8 }
```

Listing 5.2: LookupRule, simplified

5.3.2 Lookup Rule

LookupRule is the only base rule we have implemented and represents the only leaf operator (i.e. access path) we support. Lookup looks up the specified word in a given index, and returns the records. In short, it provides the following as shown in Listing 5.2:

Stats stores the **cardinality**, i.e. an estimate of the number of records that will be returned, and **tuple size**. This is important not only to determine the cost of the lookup, but also for all the consumers. A stubbed system catalog provides this information, as described in Section 4.3.2. It is fetched and set by the rule binder.

Index and **Word** being looked up, to be able to reconstruct the physical algebra.

Order is the **physical ordering**, which in turn is used to determine all logical orderings. Indexes in MARS are typically clustered on `DocumentId`.

Initialize will initialize the BasicPlanSetState for the specified PlanSet, while

UpdatePlan updates the cost, ordering and sharing of the specified plan.

The complete lookup rule is given in Section B.9.

5.3.3 Sort Rule

The sort rule is shown in Listing 5.3 and does very little, since it is only a helper rule. Its only purpose is to store the *Order* description of the sort it performs and to calculate the cost of itself inside *UpdatePlan*. Sharing is the same as its input.

It can be invoked by other rules whenever a sorted input is required, but not met by the input. Currently, it can be invoked in two situations:

1. By the Join rule, which requires the correct ordering to be able to build a MergeJoin plan.

2. By the optimizer itself which has to make sure that the optimized query satisfies the requested output ordering for the query.

As such, this is the only exception to the principle of the optimizer not knowing about the rules. As sorting is a so fundamental operator, this is not a problem.

This rule can be more intelligent about considering the degree of ordering already present in the input. For example, if the input is sorted on (a, b), but an ordering for (a, b, c) is warranted, the cost is likely to be a lot less than if the input had no ordering at all.

```
1 public class SortRule : AbstractRule, IHelperRule {
2    OrderProxy Order;
3    void UpdatePlan(Plan plan);
4 }
```

Listing 5.3: SortRule, simplified

```
1 public abstract class UnaryRule : AbstractSearchRule {
2   public override void Search(PlanSet plans, ICost limit) {
3      PlanSet input = qo.GeneratePlans((plans.Properties-Produced.BitSet) | Required[0],limit);
4      if (input == null)
5         return; // No plans
6
7      if (plans.Count == 0)
8         plans.State = CalcPlanSetState(input.State);
9
10     foreach (Plan inputPlan in input) {
11        Plan newPlan = new Plan(inputPlan);
12        UpdatePlan(newPlan);
13        plans.AddPlan(newPlan);
14     }
15   }
16   protected virtual BasicPlanSetState CalcPlanSetState(BasicPlanSetState input) {
17      return input.Filter(Selectivity);
18   }
19   void UpdatePlan(Plan plan);
20 }
```

Listing 5.4: UnaryRule, simplified

5.3.4 Unary Rule

Unary rules are rules with only one child, and most of them can be implemented quite easily. The UnaryRule provides this implementation. It implements a basic search strategy of constructing all plans where the rule itself is the topmost one, thereby producing plans with the rule in all possible locations. However, leaving the default implementation in place for all unary rules will make the search space too big (it increases exponentially with the number of rules), so care should be used.

Listing 5.4 shows the implementation of UnaryRule. The *Search* method basically asks the plan generator to produce all plans with the requested properties **minus** what the rule itself produces **plus** what it requires, effectively placing itself on the top. If no plans can be generated, nothing is done. Otherwise, if this is the first plan being produced for these properties, it sets some PlanSetState and then creates new plans with itself on the top.

5.3.5 Selection Rule

SelectionRule is shown in Listing 5.5 and constructs selections in the query. It is a unary rule and therefore inherits from UnaryRule. We have not optimized its implementation and just left the standard unary search method in place.

An important task for the rule binder is to determine the selection's `Selectivity`. This greatly influences not only the cost of the select operator, but also to any consumers above it. The cost of the select operator is also affected by its `PredicateCost`.

A problem with today's implementation is that we have very little to base the selectivity estimates on. Therefore, the selectivity of a SelectionRule is simply hard-coded to 0.1 now. How-

```
1 public class SelectionRule : UnaryRule {
2    double PredicateCost, Selectivity;
3    HashSet<Dependency> InducedDependencies;
4    ExpressionProperty SelectionFilter;
5    void UpdatePlan(Plan plan);
6 }
```

Listing 5.5: SelectionRule, simplified

```
1 public class GroupOperatorRule : UnaryRule {
2    GroupingProxy Grouping;
3    IList<NodeAttribute> GroupingFields;
4    IDictionary<string, Aggregator> Aggregators;
5    void UpdatePlan(Plan plan);
6 }
```

Listing 5.6: GroupOperatorRule, simplified

ever, we override the values in various tests to ensure that the selects are properly moved.

Another important task is to determine what functional dependencies (stored in `InducedDependencies`) are introduced by the selection, as this affects the available logical orderings as explained in Chapter 6. To do this, we utilize MARS' parser for predicate expressions, which parses it to a LINQ (Language Integrated Query, a .NET feature) expression tree. We then traverse the tree, looking for attribute references (which induces a dependency on the operator that produces it), as well as expressions like `attribute = CONST` or `attribute1 = attribute2` which introduce functional dependencies. We do not perform any predicate rewriting like DeMorgan or the like.

The complete selection rule is given in Section B.10.

5.3.6 Group Rule

GroupRule, shown in Listing 5.6, constructs group by-operators in the query. As selection, it is a unary rule and therefore inherits from UnaryRule. A group operator only lets through the attributes it groups on, `GroupingFields`, in addition to adding attributes for the aggregate functions, `Aggregators`. Dependencies are set to every operator below it and the above will be forced to stay there. This is more limiting than necessary — but we have not studied exactly when it would be interesting to move operators above a group operator.

It represents both stream and hash group as one rule (as opposed to the join rule, which uses helper rules for merge and hash joins), and `Grouping` stores the grouping of the tuple stream it desires. This is because MARS also handles them as one operator, automatically switching between the two based on ordering of the input. Interestingly, MARS prevents us from doing an optimization by doing this, as described in Section 6.2.2.

As the various MARS aggregate classes do not override *Equals* we could not easily implement *StructurallyIdentical*. However, none of our example queries reuse the output from a Group-operator, so it has not been a big deal.

5.3.7 Map Rule

Map operators are represented by the MapRule (Listing 5.7) and performs only attribute removal, rename or computation of new attributes, not duplicate removal, *distinct*, on the tuple stream. `ExpressionCost` stores the cost of the expressions (currently static), `ParameterMap` the

```
public class MapRule : UnaryRule {
    double ExpressionCost;
    Dictionary<IdentifierProperty, ExpressionProperty> ParameterMap;
    HashSet<Dependency> InducedDependencies;
    void UpdatePlan(Plan plan);
}
```

Listing 5.7: MapRule, simplified

expressions with output alias, while InducedDependencies stores the functional dependencies induced for orderings. *UpdatePlan* updates cost, ordering and sharing.

Maps are handled a bit differently from other operators. To reduce the search space, a pre-processing step removes all map operators that only rename or remove attributes from the query. Attribute renames are handled transparently by the (node, attribute)-tuple way of referencing attributes previously described. Attribute removals are also handled transparently, as the optimizer (actually, the rule for the Output operator) will insert a map operator as the topmost operator in the query if there are too many attributes, wrong names or wrong order of the attributes. Maps that produce new attributes are left as they are, but are locked in place just beneath the first operator that requires those attributes.

In many cases, it is preferable to insert a map operator before potentially disk-spilling operators like Sort and HashJoin, but we do not currently do this. More on suggested improvements for Map can be found in Section 8.5.2.

An important task of the map rule binder is to identify the attributes referred in the various expressions. At the moment, this is done by simple sub-string matching using regular expressions — for every input attribute, if the attribute name is a sub-string of an expression (with word boundary checks, so "Score" does not match "ProxScore"), a dependency on that attribute is added. Ideally, this would use the same parser as MARS does, to instead reason on the AST of the expressions.

Another important task is to register attribute renames with the optimizer, so that the optimizer can correctly identify the origin for an attribute that passes through a map operator.

5.3.8 Binary Rule

As for unary rules, we also have an abstract base class for binary rules (rules with two children). Currently, it only includes some convenience properties and propagation of sharing properties.

5.3.9 Join Rule

The JoinRule constructs joins in the query graph and is thereby responsible for one of the core problems of query optimization; join ordering. It is also the only binary rule we have implemented, apart from ScoreOccurrences which does not come even close in complexity. The most important code is shown in Listings 5.8 and 5.9. Please note that these code samples have been simplified more heavily than usual to fit here. We have removed function parameters and unimportant variables for brevity.

[Neu05] gives the implementation for bushy join enumeration only, but we have implemented left-deep enumeration as well. Which to use is controlled by the JoinEnumeration property on the join rule. Line 10 in Listing 5.8 shows the distinction between bushy and left-deep. Let us consider bushy joins first, as this is the simplest case.

The basic idea is that the rule is requested to produce a set of properties. Some of them it produces itself, and some must be requested from the input plans. The difference from unary

rules is that it can get them from either input plan. Its `ReqLeft/ReqRight` properties *must* be requested from the left or right input (for example, the attributes the join predicate references), but the rest can be freely chosen. Therefore, we determine all the properties that either of its children must satisfy (the freely chosen). This is `wantedProps` = `Requested properties` - (`Produced` | `ReqLeft` | `ReqRight`), which happens on line 9. Then *InternalSearch* is called to produce plans. It takes four parameters: 1) the `PlanSet` to add results to, 2) the freely chosen properties, 3) properties that must be on the left side, and 4) properties that must be on the right side.

For left-deep enumeration, the difference is how the freely chosen properties are determined. When the join rules are initialized, a property `OtherJoins` is set, containing the produced and required properties of all the other join rules. For left-deep plans, the right input should be a relation, so all other remaining joins should be in the left input. Therefore, `OtherJoins` is removed from the freely chosen properties on line 14. `remainingJoins` is computed as the joins remaining below this rule. The ones performed including and above this rule, closer to the root, are removed by intersecting `OtherJoins` with the requested properties and removing the produced properties for this rule. Also, `ReqLeft/ReqRight` are removed as these are requested explicitly for the left or right input.

All remaining joins are explicitly forced to the left input by specifying `ReqLeft` | `remainingJoins` as the left parameter to `InternalSearch`. *InternalSearch* needs to be called a second time with `ReqLeft/ReqRight` swapped (but with `remainingJoins` still on the left side) to enable join ordering at all — otherwise it would just force one possible order. We can do this since $A \bowtie B$ is equivalent to $B \bowtie A$. Note that this also requires swapping `OrderLeft/Right` and `SortLeft/Right` in *InternalSearch*, but we have omitted this for brevity.

The logical join rule should never build any algebras or calculate cost (this is left up to the helper rules), so *BuildAlgebra* and *Cost* just throws an exception.

```
1  public class JoinRule : BinaryLogicalRule {
2     HybridHashJoinRule HybridHashJoin; MergeJoinRule MergeJoin;
3     OrderProxy OrderLeft, OrderRight; IRule SortLeft, SortRight;
4     BitSet OtherJoins;
5     JoinEnumeration JoinEnumeration;
6     IList<NodeAttribute> JoinKey;
7
8     public override void Search(PlanSet plans) {
9        BitSet wantedProps = planSet.Props - (Produced | ReqLeft | ReqRight);
10       if (JoinEnumeration == Bushy) {
11          InternalSearch(plans, wantedProps, ReqLeft, ReqRight);
12       }
13       else if (JoinEnumeration == LeftDeep) {
14          wantedProps -= OtherJoins;
15          remainingJoins = (OtherJoins & plans.Props) - (Produced | ReqLeft | ReqRight);
16          InternalSearch(plans, wantedProps, ReqLeft | remainingJoins, ReqRight);
17          InternalSearch(plans, wantedProps, ReqRight | remainingJoins, ReqLeft);
18       }
19       // And so on for RightDeep and zig-zag.
20    }
21    public override OperatorNode BuildAlgebra(Plan plan) {
22       throw new OptimizerException("Delegated to helper rules.");
23    }
24    protected override BasicCost Cost(Plan plan) {
25       throw new OptimizerException("Delegated to helper rules.");
26    }
27 }
```

Listing 5.8: JoinRule, heavily simplified

```
1  void InternalSearch(PlanSet plans, BitSet wantedProps, BitSet left, BitSet right) {
2     foreach (BitSet leftProps in wantedProps.Walk()) {
3        PlanSet leftPlans = GeneratePlans(left | leftProps, OrderLeft, SortLeft);
4        if (leftPlans == null)
5           continue;
6
7        BitSet rightProps = wantedProps - leftProps;
8        PlanSet rightPlans = GeneratePlans(right | rightProps, OrderRight, SortRight);
9        if (rightPlans == null)
10          continue;
11
12       if (plans.Count == 0)
13          plans.State = CalculateState(leftPlans, rightPlans, Selectivity);
14
15       foreach (Plan leftPlan in leftPlans) {
16          foreach (Plan rightPlan in rightPlans) {
17             // Merge join
18             if (leftPlan.Order >= OrderLeft && rightPlan.Order >= OrderRight) {
19                Plan mergePlan = new Plan(leftPlan, rightPlan);
20                MergeJoin.UpdatePlan(mergePlan);
21                plans.AddPlan(mergePlan);
22             }
23
24             // Hybrid Hash join
25             Plan hashPlan = new Plan(leftPlan, rightPlan);
26             HybridHashJoin.UpdatePlan(hashPlan);
27             plans.AddPlan(hashPlan);
28          }
29       }
30    }
31 }
```

Listing 5.9: *InternalSearch* of JoinRule, heavily simplified

We now look at the implementation of *InternalSearch* in Listing 5.9. First, the freely chosen properties are distributed between the left and right sub plan in all possible ways, asking the plan generator to produce plans for each possible combination. This is done using the *BitSet.Walk* method on line 2 which produces all permutations of `wantedProps`. First, the rule asks for the left input plans. If there are none, we continue to the next left/right distribution. We then calculate the `rightProps` on line 7 as the remaining properties in `wantedProps` not in `leftProps` and generate the right input plans. If there are none, we continue to the next left/right distribution. If the PlanSet is empty, we then initialize its PlanSetState on line 13.

The loops on lines 15-29 iterate over all possible join plan combinations of the input plans. The JoinRule itself only represents the *logical* join — the different *physical* join algorithms are represented by helper rules. One plan is actually added for each join algorithm. First, if the input plans are ordered on the join keys, a plan for merge join is created and then a plan for hash join is always created. This is done by calling *UpdatePlan* on the corresponding helper rule, giving it the newly created plan as parameter. Dominated plans are pruned automatically by PlanSet.

HybridHashJoin- and MergeJoin Rules

The HybridHashJoinRule and MergeJoinRule are helper rules. They are not directly used by the optimizer, but consulted by the JoinRule. Therefore, they do not implement the *Search* method, but only *UpdatePlan* and *BuildAlgebra*. *UpdatePlan* is responsible for updating the costs.

```
1  public class ScoreOccurrencesRule : AbstractSearchRule {
2     // Various operator parameters.
3
4     override void Search(PlanSet planSet, ICost limit); {
5        if (planSet.Properties != Produced ∪⋃Required)
6           return;
7
8        foreach (BitSet required in Required)
9           PlanSet plans = qo.GeneratePlans(required, limit);
10       // ... Find best combination of input plans and create a new plan.
11    }
12    void UpdatePlan(Plan plan);
13 }
```

Listing 5.10: ScoreOccurrencesRule, simplified

5.3.10 ScoreOccurrences Rule

ScoreOccurrences takes its inputs from one or more Lookups or ONear/Nears. Based on the number of occurrences, how close they are to each other, etc. the operator assigns one or more score-attributes to the document.

It does not make much sense to attempt to move this operator around, so *SetDependencies* locks the operator in place. This can also be seen in the *Search* method in Listing 5.10. If it is tried used to construct something different that exactly itself and what it requires, that search branch is aborted. Also, the sub-plans searched for are exactly the properties required. We have omitted the rest of *Search()* for brevity. *UpdatePlans* updates cost, order and sharing.

As with Group, We could not properly implement *StructurallyIdentical* without resorting to ugly hacks, as the various "Scorers" in MARS do not override *Equals* et al.

6 Orderings and Groupings

"Anything that happens, happens. Anything that, in happening, causes something else to happen causes something else to happen. Anything that, in happening, causes itself to happen happens again. All of this, however, doesn't necessarily happen in chronological order."

— Douglas Adams (Mostly Harmless)

6.1 Introduction

Some of the important operators found in MARS, for example Join and Group, include at least two physical implementations — one that operates on sorted or grouped input and one that does not. For Join, MergeJoin directly merges two sorted inputs, while HybridHashJoin has to be used if the inputs are not sorted on the join key. Actually, HybridHashJoin can take some advantage of a sorted left input, but this is up to the cost model to recognize. For Group, StreamingGroup groups and aggregates a sorted (or grouped, as we will explain) input with minimal buffering, while HashGroup accepts any input ordering/grouping, but at the cost of significant more memory consumption and CPU usage. (Streaming- and HashGroup is actually realized as the same operator in MARS.)

The point is that the variant working with sorted or grouped input allows for much cheaper evaluation, usually both in terms of CPU-, memory- and IO-cost. It is therefore preferable to have the optimizer schedule for instance MergeJoin or StreamingGroup whenever beneficial. Of course, one can always use these operators by inserting a sort operator right before it, but the result is possibly an even more expensive plan than using the non-ordered operator in the first place. One middle case is a join with one of the inputs already sorted on the join key. The key point is therefore that the optimizer should be able to *recognize data already ordered* in the required way and schedule the cheaper operator variants. The cost model also needs to recognize the case where HybridHashJoin is slightly cheaper if given a sorted left input. Since the plan generator possibly considers millions of plans, this recognition needs to be very fast.

In addition to different physical operators requiring ordered input, the query may also specify that the output result set should be sorted. This can always be solved by inserting a sort operator just before the *output* operator, but doing so if the query happens to already provide the data in the requested ordering is redundant work.

To support all this, the optimizer needs to be able to reason about available orderings and groupings when combining smaller sub plans into larger plans using operators with different order and grouping requirements, and ultimately the complete query. Different orderings or groupings can originate from clustered (ordered) index scans, sorts or group operators.

Of course, none of this is specific to our optimizer — all query optimizers have to deal with

this. It is particularly interesting for MARS, since most full-text indexes are clustered (ordered) by *document id* and *word position*. This can be exploited using merge joins, resulting in great improvements in query evaluation time.

We have looked at two different ordering frameworks to add this feature to the optimizer, [SSM96] and [NM04], and found that both would fit our needs. However, the latter one seemed to be more promising since it in addition to orderings also could handle groupings, while claiming to be significantly faster than the former. Our implementation is therefore largely based on the description in [NM04], mostly using the algorithms described there. In this chapter we therefore give an introduction to the framework, but refer the reader to the source for a deeper understanding and detailed description of the algorithms involved.

6.2 Overview

The logic the optimizer needs to reason about orderings and groupings is separated into its own component, the *Order Manager*. It encapsulates the algorithms and data structures used and provides a clean API to the optimizer. More specifically, the order manager needs to provide the following services:

1. Keep track of the available orderings and groupings for a plan at all times without much overhead.

2. Offer an API to check if a plan satisfies a specific ordering or grouping. This is utilized by the plan generator when reasoning about what physical operators to schedule.

3. Initialize a plan with a certain ordering or grouping, for instance from a clustered index scan, sort or group operator.

4. Offer an API for operators/rules to declare how they affect orderings/groupings. For example, applying the selection $a = 50$ will cause the data to be ordered by a.

There is more to orderings and groupings for a tuple stream than one might expect and we now continue by introducing some important concepts.

6.2.1 Orderings

The overall goal of keeping track of orderings is to avoid inserting redundant sort operators. The plan generator will actually always try to construct a plan with for instance merge joins, but the cost model will reject those plans if the cost of sorting first becomes greater than using a hash join. It therefore needs to keep track of the relevant orderings for query optimization, the *interesting orderings* [SAC$^+$79]:

1. All orderings required by a physical operator that may be used in the query, including the requested ordering(s) of the query result set.

2. All orderings produced by a physical operator that may be used in the query, including index scans and sort operators.

These orderings are *logical orderings* in that they describe logical orderings the tuple stream must satisfy, as opposed to *physical orderings* which is the actual ordering of the tuples. A tuple stream will only have one physical ordering, but can satisfy several logical orderings. For example, given the schema (a, b), the tuple stream $((1, 1), (3, 3))$ has one physical ordering, the actual stream, while is satisfies the logical orderings a, b, ab and ba.

Operators like Sort and Lookup affect the physical ordering since they produce or alter an actual order of the tuples. `Sort[a]` will make the tuple stream satisfy the logical ordering a by rearranging the tuples. Other operators, like Select and Map/Projection affect the logical ordering. Given the schema (a, b), applying `Select[b=50]` to a tuple stream satisfying the ordering (a), will make it satisfy the orderings a, b, ab and ba. *Functional dependencies* (FDs) are used to express such deductions. Note that a tuple stream satisfies all prefixes that can be constructed from the orderings it satisfies. A tuple stream satisfying ab will always satisfy a.

Given a functional dependency $b \rightarrow c$, and a logical ordering containing b, the FD can be used to deduce new logical orderings by inserting c at any position after the position of b. To see this, consider that the FD says that "for equal values of b, c will be equal as well". Therefore, if a tuple stream is ordered by b, we can be sure that it is ordered by bc, since for each distinct value of b, c will have the some constant value. For a functional dependency on the form $ab \rightarrow c$, both a and b need to be present and c can be inserted after the position of whichever is last. For a dependency $\rightarrow c$ (which means that c is constant), c can be inserted at all locations. This is exactly what happened in the `select[b=50]` example. For equalities, $b = c$, b can be replaced by c and vice versa, in addition to being inserted as described above.

6.2.2 Groupings

In addition to keeping track of available logical orderings, if the query contains group operators, it may be beneficial to keep track of available groupings as well. For example, when applying a group operator to a tuple stream that is already grouped, one can use a streaming group instead of a hash-based group, saving memory and CPU cycles.

Just as for orderings, we have a set of *interesting groupings*:

1. All groupings required by a physical operator that may be used in the query.

2. All groupings produced by a physical operator that may be used in the query.

The reason for keeping track of groupings separately from orderings is that a grouping does not imply an ordering. For instance, given the schema (a, b), the tuple stream $((2, 3), (1, 2), (2, 4))$ is grouped by $\{a, b\}$ but not ordered by anything. However, an ordering always implies the corresponding grouping, i.e a tuple stream ordered by (a, b) satisfies the groupings $\{a\}$ and $\{a, b\}$. Note that while an ordering is given as a *sequence* of attributes, a grouping is given as a *set*. Therefore the prefix property that holds for orderings does not hold for groupings. A tuple streamed grouped by $\{a, b\}$ need not be grouped by $\{a\}$, and this is also true for the example tuple stream above.

Given a functional dependency $b \rightarrow c$, and a logical grouping containing b, the FD can be used to deduce a new logical grouping by adding c to the set of attributes. For a functional dependency on the form $ab \rightarrow c$, both a and b need to be present in the grouping, while for $\rightarrow c$, c can just be added to the set. For equalities, $b = c$, b can be replaced by c and vice versa, in addition to being added to the set.

Currently, we do not actually need the groupings functionality, as this is more than MARS can currently make use of. More specifically, the group operator in MARS is designed to automatically choose between streaming group and hash group based on the nature of the input stream. Currently, it only checks if grouping attributes is a prefix of the ordering of the input tuple stream. This always guarantees correct results, but is misses the case where the tuple stream is only grouped, not sorted. However, it should be easy for *fast* to extend their group operator with the functionality to recognize that a tuple stream is grouped, not sorted, for instance by being told so by our optimizer. Therefore we still implemented this functionality.

6.2.3 Functional Dependencies

As previously mentioned, *functional dependencies* (FDs) are the way rules/operators express deductions for orderings and groupings. More specifically, how an operator influences the logical orderings and groupings is expressed as a set of FDs it applies to the tuple stream when it is performed. These FDs are then used by the ordering component to deduce a new and wider set of logical orderings and groupings. Note that a Sort operator does not apply FDs, instead it resets the ordering state with the new physical ordering it produces. Any applied FDs will still hold and will need to be reapplied after the sort.

We now list the major sources of functional dependencies and give examples.

Key constraints FDs typically arise from primary keys. For a document relation, one will typically have the FD `DocumentId` → `DocumentName, DocumentType`. For example, this means that if the tuple stream is ordered on `DocumentId`, it will be ordered on `DocumentId, DocumentName, DocumentType` as well.

Join predicates typically lead to FDs on the form $a = b$ (for equality joins).

Filter predicates on the form $a = b$ gives the same FD as above. A filter predicate on the form $a = constant$ will give the FD $\rightarrow a$.

Map/projection expressions. A projection $d = f(a, b, c)$, for example $d = a + b \times c$, results in the FD $abc \rightarrow d$, since for specific values of abc, d will always have the same value.

6.2.4 Orderings and Groupings for MARS' Operators

We now specify the orderings and groupings produced and required by the different operators in MARS, as well as the functional dependencies they induce. They are shown in Table 6.1.

Lookup requires no order, but produces an order depending on the index schema. For instance, a full-text lookup will be ordered by `DocumentId, Position`. For most lookups, `DocumentId` is the primary key. This gives the FDs `DocumentId` → rest.

For ScoreOccurrences and ONear/Near, $\vec{a}$ contains the document grouping field for the inputs, most commonly `DocumentId`. $\vec{b}$ contains the word position field for the inputs, most commonly `Position`. They both require all inputs to be ordered by the grouping and position field and will let this ordering through, in addition to inducing an equivalence between all grouping attributes, much like a join would do.

Map and Select only induce dependencies, they do not alter them.

MergeJoin (equi-join) can join an unlimited number of tuple streams, where $\vec{a}, \vec{b}$ and so on represent the join keys. It requires ordering on the join keys for all inputs and will not destroy this ordering. It will, just as HybridHashJoin, induce equivalence between the join keys, but HybridHashJoin does not require or produce any ordering.

A Group on $a_1, .., a_n$ does not require any grouping (although it will run more efficiently if the input is already grouped on $a_1, .., a_n$), and produces a grouping on $a_1, .., a_n$.

A Sort on $a_1, .., a_n$ alters the physical ordering and it therefore produces the ordering $a_1, .., a_n$. Trim does not affect ordering at all.

Finally, for the output operator o_1 and so on represents the requested output orderings for the (multi) query. It therefore requires these orderings.

6.2.5 Plan Generator Interface

The ordering component offers an abstract data type (class) OrderingState, which represents a set of all the logical orderings and groupings available as well as the set of FDs applied. The

Operator	Required	Produced	Induced FDs
Lookup(I)	-	$O(I)$	Keys
ScoreOccurrences$(S_1..S_n, \vec{a}, \vec{b})$	$\{a_1, b_1\} \in O(S_1)$ .. $, \{a_n, b_n\} \in O(S_n)$	$O(S_1)$	$a_1 = .. = a_n$
ONear/Near$(S_1..S_n, \vec{a}, \vec{b})$	$\{a_1, b_1\} \in O(S_1)$ .. $\{a_n, b_n\} \in O(S_n)$	$O(S_1)$	$a_1 = .. = a_n$
Map$(S, a = f(b, c))$	-	-	$bc \rightarrow a$
Select$(S, a = b)$	-	-	$a = b$
MergeJoin$(S_1..S_n, \vec{a} = .. = \vec{z})$	$\vec{a} \in O(S_1) \wedge .. \wedge \vec{z} \in O(S_n)$	$O(S_1)$	$\vec{a} = .. = \vec{z}$
HybridHashJoin$(S_1, S_2, \vec{a} = \vec{b})$	-	-	$\vec{a} = \vec{b}$
Group$(S, a_1..a_n)$	-	$\{a_1, .., a_n\}$	-
Sort$(S, a_1..a_n)$	-	$(a_1, .., a_n)$	-
Trim(S)	-	-	-
Output$(S_1..S_n, o_1..o_n)$	$o_1 \in O(S_1)..o_n \in O(S_n)$	-	-

$O(S)$	The logical orderings available for tuple stream S.
$(a_1, ...a_n)$	The logical or physical ordering $a_1..a_n$.
$\{a_1, ..., a_n\}$	The logical grouping $a_1, .., a_n$.

Table 6.1: Required and produced orderings and induced functional dependencies

plan generator annotates all plans generated with this data type, so operators like MergeJoin can easily check whether a plan satisfies its orderings requirements or not. Since the number of available orderings increases very rapidly when applying functional dependencies, storing these naïvely is not an option. Orderings and groupings are stored by wrapping the Order class, while FDs are stored as a bit mask (more on this later).

The ordering components offers the following API functions for OrderingState to the plan generator:

- *OrderingState OrderManager::GetOrdering(OrderDescription) / GetGrouping(GroupingDescription)*: When a plan is constructed from an operator that produces a physical ordering or grouping, such as a Sort, Group or Lookup (index scan), the plan generator needs to initialize the ordering state to this ordering.

- *OrderingState OrderingState::Apply(Set<FD>>)*: Whenever a operator is added to an existing plan, the plan generator needs to apply the FDs it induces to deduce the new order and grouping state.

- *OrderingState OrderingState::Apply(OrderingState)*: If an operator with more than one input is applied, the plan generator must merge the ordering states and the set of applied FDs from the two inputs.

- *bool OrderingState::Satisfies(Order)/Satisfies(Grouping)*: Used by the optimizer to check if a plan satisfies the ordering or grouping requirement for an operator, for instance a Merge-Join. If it does, the plan generator will consider the new plan with, for instance, Merge-Join.

6.3 State Machine Model and Implementation

6.3.1 Example Query

We start by introducing a query that will illustrate how the plan generator uses the ordering and grouping component, as well as how the component works internally. The query is shown in Figure 6.1. It may seem somewhat artificial, but we have constructed it to be able to illustrate all the concepts. Note that all figures in this chapter have been somewhat simplified to emphasize the important concepts.

The Lookups produces the attributes `DocTypeId`, `Extension`, `DocId`, and `DocName`. It is an imagined materialization of a join between documents and their types, where `DocTypeId` is the id of the document type, while `Extension` is the file extension for the type, for instance `.doc`. It is clustered (ordered) by `DocTypeId` ascending. To avoid name clashes, we have prefixed `DocTypeId` with 0_ and 1_ for the different branches.

In the left branch, the Group operator groups the documents by `DocTypeId`, aggregating the count of each type. Then, a Map uses an imagined *TypeDescription* function to look up the description for a document type, for instance "MS Word Document". Next, another group is applied (no point, just for the sake of illustrating a concept), before the result set is trimmed to the first 10 tuples, ordered by (`SecondCount DESC`, `DocTypeId`). Finally, another sort is applied to get the final output order of (`SecondCount DESC`, `DocTypeId`, `DocTypeDesc`). We have also included a query branch that only reads the Lookup results sorted to illustrate how share equivalence is handled later.

The next thing that happens (in the context of orderings) is that the query is fed through the OrderingExtractor and SortRemover pre-processors. This would result in the Sort operators being removed and the Output and Trim operators being annotated with the orderings they require, as extracted from the sort operators. This is shown in Figure 6.2.

We can now start looking at the set of interesting orders and groupings and functional dependencies for the query:

Interesting orderings	
(0_DocTypeId)	Produced by the right Lookup and required by the Output operator.
(1_DocTypeId)	Produced by the left Lookup.
(SecondCount, 1_DocTypeId)	Required by the Trim operator.
(SecondCount, 1_DocTypeId, DocTypeDesc)	Required by the Output operator.

Interesting groupings	
{1_DocTypeId}	Required by the lower Group operator.
{DocTypeDesc, 1_DocTypeId}	Required by the upper Group operator.

Functional dependencies	
DocId → DocTypeId, Extension, DocName	DocId is the primary key of the lookup relation.
1_DocTypeId → DocTypeDesc	Induced by the Map operator.

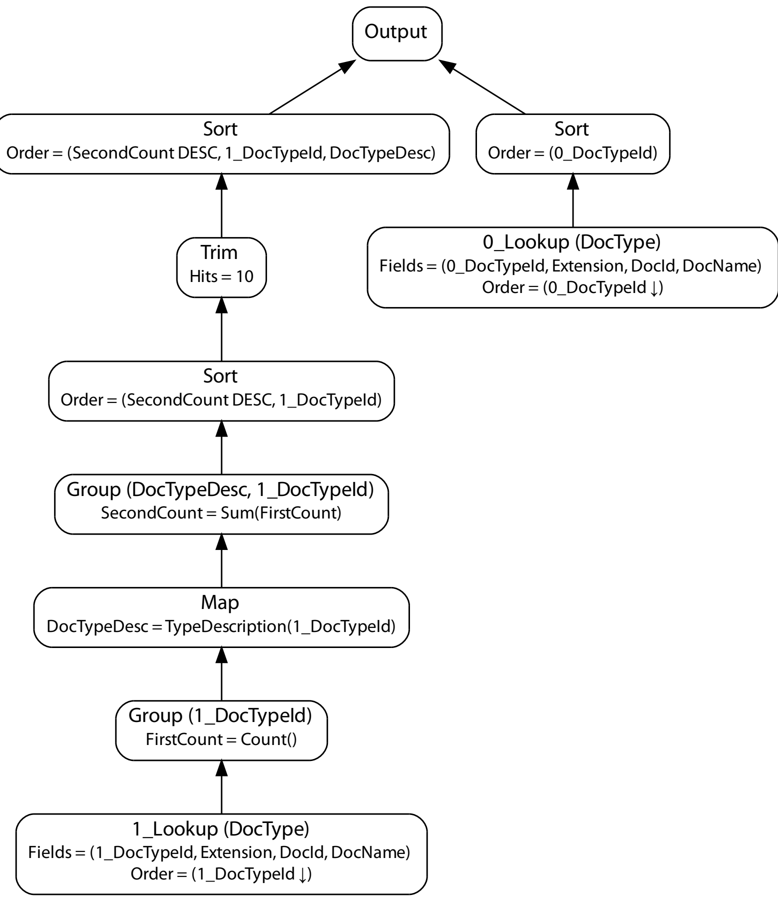


Figure 6.1: **Input** sample query for orderings and groupings

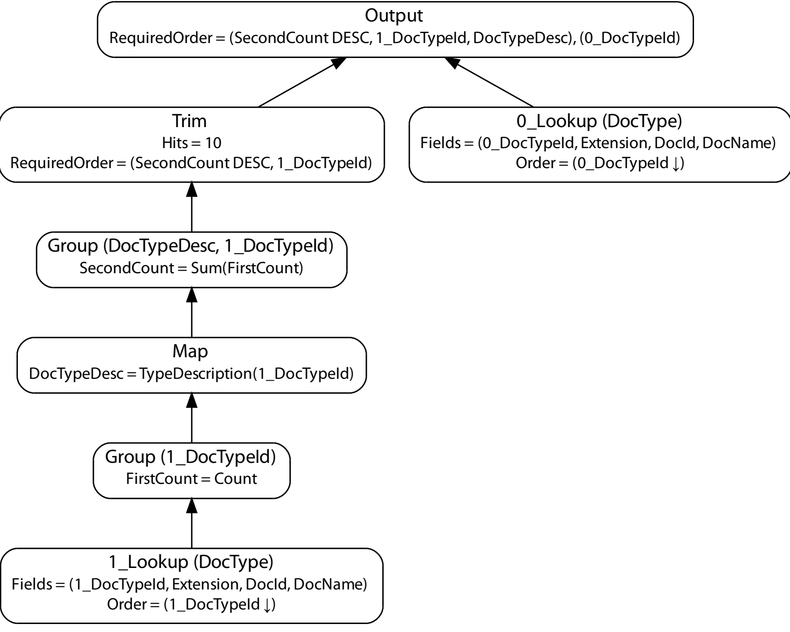


Figure 6.2: After OrderingExtractor run

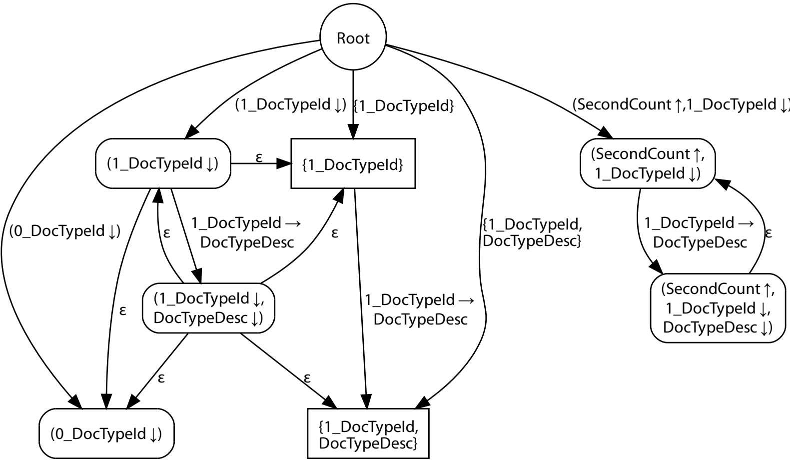


Figure 6.3: Finished NFSM for the sample query

6.3.2 Overview

As previously mentioned, having the `Order` class explicitly maintain the set of logical orderings and groupings is prohibitively expensive. However, the key observation is that it does not need to offer access to this set, but rather an API to test if a certain ordering or grouping is a member of the set. This allows for optimizations. The idea from [NM04] is to represent logical orderings and groupings as states in a *finite state machine* (FSM). The edges (transitions) in the FSM are labeled by functional dependencies. As such, with the FSM in a state representing a set of logical orderings and groupings, applying a functional dependency brings the FSM to another state, representing the new (and larger) set of available orderings and groupings deduced using this dependency. The exceptions are the edges from the *root* node, which is used to initialize the FSM with a physical ordering or grouping (after a `Sort`-operator, for instance). Therefore, the set of available orderings and groupings for all states can be pre-computed, allowing for fast lookup and efficient storage in bit masks.

Before we get as far as described above, we need to go trough several steps. We present the idea in this overview section and then go into the detailed construction in the next sections. First, the sets of interesting orderings, groupings and functional dependencies are used to construct a *non-deterministic finite state machine* (NFSM), which can be seen in Figure 6.3.

On this NFSM, each state (except the root) represent *one* logical ordering or grouping. Note that orderings are annotated with (...) and rounded nodes, groupings with {...} and square nodes and functional dependencies ... → .... For example, by following the `(1_DocTypeId)` edge from the root, we end up with logical ordering `(1_DocTypeId)`. The right ϵ edge means that the node which represents the grouping `{1_DocTypeId}` is directly available, since an ordering implies a grouping on the same attributes. If we follow the edge labeled `1_DocTypeId` → `DocTypeDesc` (represents applying this FD), we arrive at the logical ordering `(1_DocTypeId, DocTypeDesc)`. This node has ϵ edges back where we came from (since an ordering on ab is also an ordering on a), and to the available groupings.

Note that the FD `DocId` → `DocTypeId, Extension, DocName` has been filtered away since no interesting ordering on `DocId`, `Extension` or `DocName` exists.

When the FSM has been constructed, each state can be mapped to the state of the `Order` ADT. Applying functional dependencies induced by operators is performed by following the corresponding edge, and the set of available orderings and groupings can be deduced by fol-

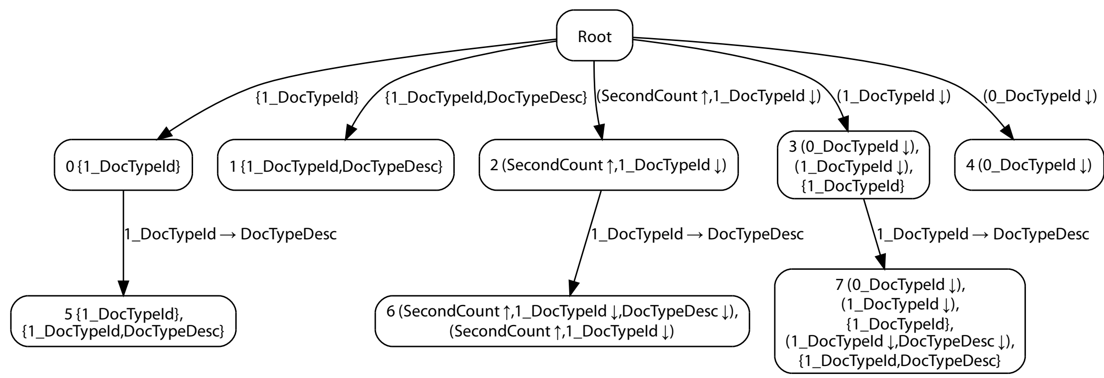


Figure 6.4: Finished DFSM for the sample query

lowing the ϵ edges. However, the non-determinism is a problem when applying functional dependencies, so we will need to convert the FSM to a *deterministic finite state machine* (DFSM).

The resulting DFSM is shown in Figure 6.4. The ϵ edges have been removed and the states now represent *sets* of logical orderings and groupings. The example from the NFSM is found again as the next rightmost branch. By following the (1_DocTypeId) edge from the root, we end up with logical ordering (1_DocTypeId) (and (0_DocTypeId), as we will see later) and grouping {1_DocTypeId}, and by following 1_DocTypeId → DocTypeDesc, we get to the additional orderings and groupings reachable by ϵ edges in the NFSM.

This DFSM is directly usable during query optimization since it is deterministic and pre-computed. By pre-computing the FSM, **checking availability for an ordering or grouping is** $\mathcal{O}(1)$, while **applying an FD is at worst** $\mathcal{O}(n)$, but this is not a huge problem since iterative reapplying rarely happens.

We now present the optimized version (Figure 6.5) of the query given in Figure 6.1 and explain how the DFSM is used.

First, the plan generator realizes that the Lookup produces an ordering and annotates its base plan with DFSM state 3 by following the edge from the root. Next, the rule for the Group operator will ask the OrderingState annotated for the Lookup plan if it has the logical grouping {1_DocTypeId}, which it has, so a Streaming Group (as opposed to a Hash Group) will be used. No FDs or physical reordering happens, so the ordering state remains the same. Then, the Map operator is applied and the ordering state of its plan will be set by applying 1_DocTypeId → DocTypeDesc to its child's ordering state. This brings us to DFSM state 7. As with the first Group, the second Group's grouping requirement is also satisfied, so a Streaming Group is used.

However, when it is time to insert the Trim operator from Figure 6.2, its ordering requirement is not satisfied, so a Sort operator, which is later merged with the Trim operator to a Sort-Trim. The DFSM is re-initialized to state 2 and the FD already applied is reapplied, bringing it to DFSM state 6. This state also contains the logical ordering required by the Output operator, so another Sort is not required.

The right query branch, is a direct read of the Lookup with required ordering (0_DocTypeId). Even though the Lookup produces the ordering (1_DocTypeId), no Sort is needed, because 0_DocTypeId is share equivalent to 1_DocTypeId.

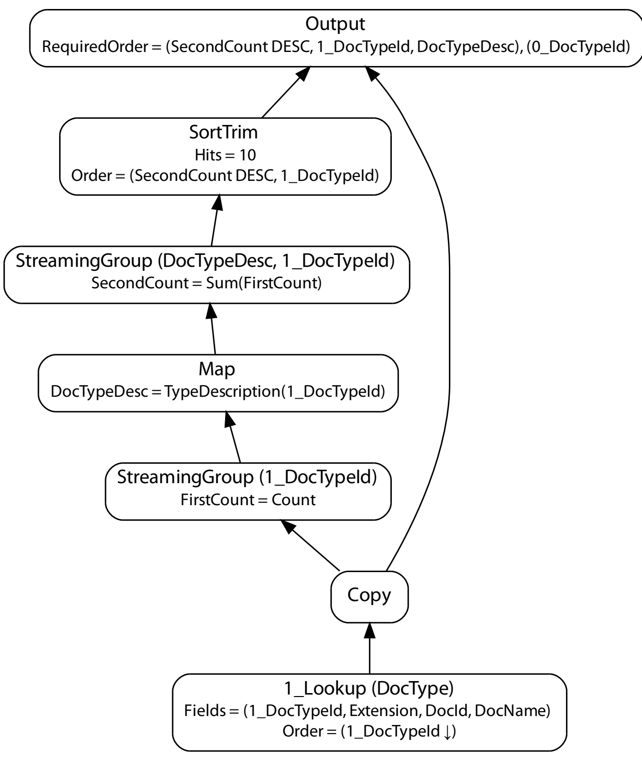


Figure 6.5: Optimized sample query

6.3.3 FSM Construction

The FSM construction consists of several steps, which we shortly introduce. The code orchestrating it can be found in Section B.11.

1. Determining Input

Before constructing the FSM, all interesting orderings, groupings and functional dependencies need to be determined. The OrderManager therefore provides an API where the rules can ask for a specific ordering or grouping which they want to use, for instance for checking their ordering requirements later. In much the same way as BitSets are created, they cannot get the ordering or grouping itself, since it has yet to be constructed by the order manager. Instead they will get an OrderProxy or GroupingProxy which the order manager keeps a reference to. Later, when the FSM has been constructed and orders and groupings mapped to an FSM state, it will fill the proxies.

2. Add States to the NFSM

By using the input gathered in step 1, the states in the NFSM are now constructed, one state for each logical or physical ordering or grouping. This includes the root state with the initialization edges between it and the states representing the physical orderings or groupings produced in the query. Figure 6.6 shows the NFSM after this step.

As can be seen on the figure, this step also includes adding ϵ edges for ordering prefixes.

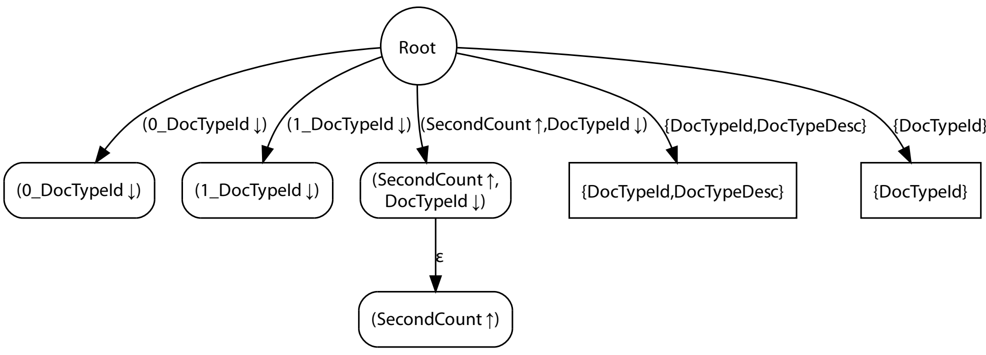


Figure 6.6: NFSM after inserting states for interesting orders and groupings

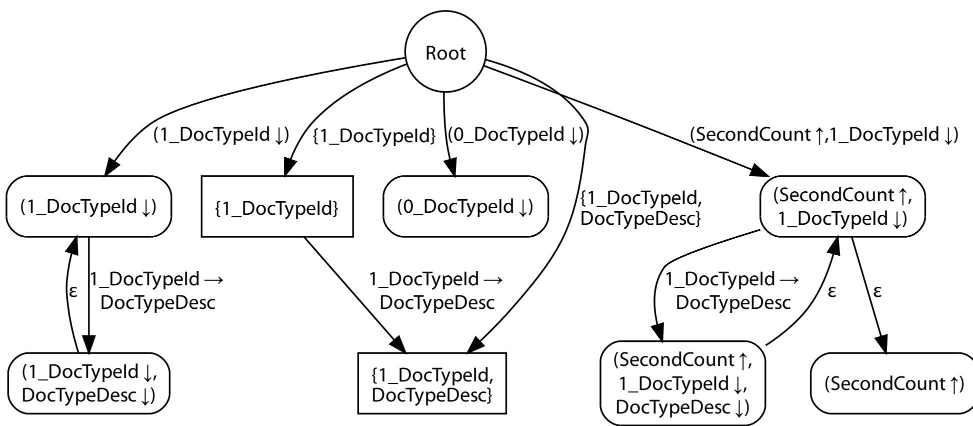


Figure 6.7: NFSM after inserting transition edges

3. Add Transition Edges to the NFSM

According to the principles described in Section 6.2, functional dependencies are now used to deduce new orderings and groupings. We will not thoroughly describe how this is done, since it is quite involved, but in short it works as follows: Each state in the NFSM is used as a starting point. Then, each functional dependency is applied to this ordering or grouping to deduce new, possibly interesting, orderings and groupings. If the new ordering or grouping is interesting (or may become interesting by applying further FDs), it is added as a new state, with an edge labeled with the FD considered leading to it from the current considered state. This is done in an iterative fashion, as new states being added also should have the different FDs considered for application. Edges from orderings to the corresponding groupings are also added.

The result can be seen in Figure 6.7.

4. Add Share Equivalency Edges to the NFSM

When rules/operators are determined to be share-equivalent, their attributes are also share-equivalent. This has importance when the equivalence class representative is used in a plan

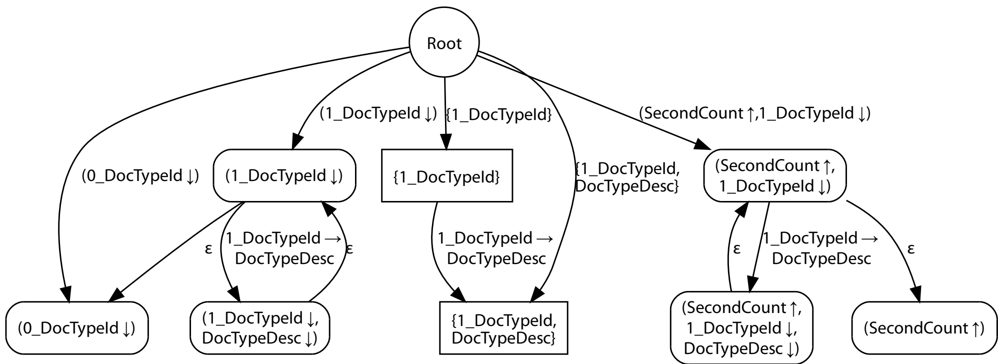


Figure 6.8: NFSM after adding share equivalency edges

instead of what is represents. For example, if `Lookup` A is share equivalent to another `Lookup` B, the ordering manager needs to recognize that an ordering on $A.a$ is also an ordering on $B.a$.

First, a share equivalency map is constructed by the plan generator that maps share equivalent attributes to their representative. This map is then used by the order manager to add ϵ edges between share equivalent orderings and groupings. In short, for each state in the NFSM, each attribute in the ordering or grouping for that state is sent through the equivalency map, constructing a new temporary ordering or grouping using only share equivalency representatives. If this new ordering or grouping is found as a state in the NFSM, an ϵ edge is added *from* this state *to* the state being considered. This is because it is the share equivalent attribute representatives that is used in place of their share equivalent counterparts in plan generation, not the opposite. The result can be seen in Figure 6.8, where an ϵ edge has been added from `(1_DocTypeId)` to `(0_DocTypeId)` since `1_Lookup` is the representative.

5. Add ϵ Edges and Optimize the NFSM

Then, ϵ edges are added between orderings and their corresponding groupings. For instance, an ϵ edge is added from the ordering (a) to the grouping $\{a\}$, as every ordering is also a grouping. This step also involves adding transitive edges, i.e. if $a \rightarrow b, b \rightarrow c$, then add $a \rightarrow c$. [NM04] also describes several techniques for reducing the size of the NFSM before constructing the DFSM. Summarized:

1. Artificial nodes that behaves exactly the same are merged.

2. Transitive ϵ transitions are optimized.

3. The length of the interesting orderings and groupings are taken into account.

We have implemented most of these optimizations, but refer the reader to [NM04] for details. The result of these two steps can be seen in Figure 6.3.

6. Convert the NFSM to a DFSM

This is carried out using the standard power set construction algorithm for converting an NFA into a DFA [LP97]. It preserves the root state and its edges. The completed DFSM is shown in Figure 6.4.

7. Pre-compute Values

The final step is to make use of the DFSM created by creating data structures for efficient storage and lookup. A transition matrix is created to represent the transitions (edges) between the states in the DFSM. Given a current state, applying a FD or an initial ordering/grouping will yield a resulting state or (-) if there is no such transition. It is implemented as a zero-index array of pointers stored in each state and allows for $\mathcal{O}(1)$ lookup. It is shown for our example query in Table 6.2.

State	Root	0	1	2	3	4	5	6	7
$(0_DocTypeId)$	4	-	-	-	-	-	-	-	-
$(1_DocTypeId)$	3	-	-	-	-	-	-	-	-
$(SecondCount, 1_DocTypeId)$	2	-	-	-	-	-	-	-	-
$\{1_DocTypeId\}$	0	-	-	-	-	-	-	-	-
$\{1_DocTypeId, DocTypeDesc\}$	1	-	-	-	-	-	-	-	-
$1_DocTypeId \rightarrow DocTypeDesc$	-	5	-	6	7	-	-	-	-

Table 6.2: Transition matrix for the DFSM

A compatibility matrix stores which DFSM states are compatible (i.e. contains) with which logical orderings or groupings. It is implemented as a bit vector stored in each state and allows for $\mathcal{O}(1)$ lookup. It is shown for our example query in Table 6.3.

State	0	1	2	3	4	5	6	7
$(0_DocTypeId)$	0	0	0	1	1	0	0	1
$(1_DocTypeId)$	0	0	0	1	0	0	0	1
$(1_DocTypeId, DocTypeDesc)$	0	0	0	0	0	0	0	1
$(SecondCount, 1_DocTypeId)$	0	0	1	0	0	0	1	0
$(SecondCount, 1_DocTypeId, DocTypeDesc)$	0	0	0	0	0	0	1	0
$\{1_DocTypeId\}$	1	0	0	1	0	1	0	1
$\{1_DocTypeId, DocTypeDesc\}$	0	1	0	0	0	1	0	1

Table 6.3: Compatibility matrix for the DFSM

Each state in the DFSM is represented by an `Order` object which stores the matrices described above for each state and additionally has an API for comparing orders and applying FDs. It is shown in Listing 6.1. The `Compatibility` member stores the compatibility matrix as a bit vector, while the `Transitions` member stores the transitions matrix. `PossibleTransitions` stores the possible transitions from this node as a bit mask.

The <= operator allows for checking compatibility between orderings (for example to check if a plan satisfies the ordering requirements of a merge join). It does a lookup in the compatibility matrix. The *Apply* method is used to apply an FD and does a lookup in the transitions matrix.

6.3.4 Plan Generator Use — OrderingState

[NM04] describes using an *abstract data type* OrderingGrouping to store the set of available logical orderings and groupings for a plan. They do not, however, specify how to store the set of applied functional dependencies. This is needed when, for instance, merging two ordering states for an operator with more than one input. One way to do this would be to tag the DFA state, but this would give 2^n states, which is prohibitive.

```
1  public class Order {
2     public BitArray Compatibility { get; set; }
3     public Order[] Transitions { get; set; }
4     public int PossibleTransitions { get; set; }
5     public int Id { get; set; }
6
7     public static bool operator <=(Order a, Order b) {
8        return b.Compatibility[a.Id];
9     }
10    public Order Apply(Dependency dependency) {
11       return Transitions[dependency.Id];
12    }
13 }
```

Listing 6.1: Order class (simplified)

```
1  public class OrderingState {
2     private int appliedDependencies; // FD bit mask, limited to 32 dependencies for now.
3     private Order order;
4     private Proxy proxy;
5     public Order Order { get { return order ?? proxy.Order; } }
6
7     public OrderingState Apply(Dependency dependency) { ... }
8     public OrderingState Apply(OrderingState other) { ... }
9  }
```

Listing 6.2: OrderingState implementation (simplified).

We have created a class OrderingState which wraps Order (our equivalent to OrderingGrouping) and stores the set of applied FDs very efficiently as a bit mask, where the nth bit represents if the FD with id n has been applied. It also handles the API logic described in Section 6.2.5. Its implementation is shown in Listing 6.2.

Currently, the FD bit mask is stored as an `int`, which means it is limited to 32 FDs. This can easily be overcome by using an array as has been done for BitSet. However, for all the queries we will encounter, 32 is more than enough. The Order property will automatically return the wrapped Order, either if set directly (for a plan, for instance), or from the proxy (for a rule, for instance)[1]. The *Apply* methods allows for applying FDs or merging two ordering states.

6.3.5 Improvements Made From [NM04]

During the implementation we found some issues with the framework from [NM04] which we worked our way around. We also claim some improvements and new features.

Share Equivalent Attributes

This is not handled at all by [NM04] or [Neu05]. Our solution is described in step 4. in Section 6.3.3.

Ordering State for a Plan

Keeping track of applied functional dependencies and ordering state merging are not described in [NM04] or [Neu05]. Our solution is described in Section 6.3.4.

[1] `a ?? b` will return `b` if `a` is null.

```
1  public OrderingState Apply(Dependency dependency) {
2     int newAppliedDependencies = appliedDependencies | 1 << (dependency.Id - 1);
3
4     Order newOrder = IterativelyApplyOrder(Order, newAppliedDependencies);
5     return new OrderingState(orderManager, newAppliedDependencies, newOrder);
6  }
```

Listing 6.3: Applying a functional dependency.

```
1  private Order IterativelyApplyOrder(Order input, int dependencies) {
2     Order oldOrder = null;
3     while (oldOrder != input) {
4        oldOrder = input;
5        int applicableDependencies = dependencies & input.PossibleTransitions;
6        int dependencyToApplyMask = applicableDependencies & -applicableDependencies;
7
8        if (dependencyToApplyMask > 0)
9           input = input.Apply(LOG2MAP[dependencyToApplyMask]);
10    }
11    return input;
12 }
```

Listing 6.4: Iterative appliance of FDs

Transitive Functional Dependencies

We encountered a problem if functional dependencies is to be exploited transitively. For example, given FD1: $a = b$ and FD2: $b \rightarrow c$, this transitively yields $a \rightarrow c$. However, to our understanding, this is not recognized by the algorithms described in [NM04], and will yield a DFSM where FD1 has to be applied twice to arrive at all orderings and groupings deducible from $a \rightarrow c$. While the best solution would be to keep track of such transitivities during NFSM generation, this would be quite complex, according to Dr. Neumann himself. As an alternative, we have implemented an iterative appliance algorithm for FDs which also solves it, at the cost of being slightly more expensive during plan generation.

Optimized, Iterative Appliance of FDs

We include the iterative algorithm described above as we believe it has some clever parts. OrderingState keeps track of the FDs applied and whenever ordering states are merged or an FD is applied, it considers each applied FD (up to this point, not only the one(s) being applied now) to see if it yields a DFSM state change, using the transition matrix previously described. This is shown in Listing 6.3. Here, we see that a new FD bit mask is calculated as the union between the existing FD mask and 1 bit-shifted to represent the Id of the new FD. Calculation of the new order DFSM state is delegated to an external method, and finally a new OrderingState is returned.

IterativelyApplyOrder considers each FD for appliance. At first, this was implemented by looping over every bit, but since this is in a performance critical path, we wanted to make sure it runs as efficiently as possible. We therefore reduced the problem to the bit twiddling shown in Listing 6.4.

The input parameters are the current DFSM state and the FDs to be considered. The outer while loop continues the process until no further transitions are possible. Note that there can be no cycles in the DFSM, since we always transition to a "richer" ordering state, so we do not need to check for this. Then the intersection between the FDs to be considered and the

possible transitions from the current DFSM state is calculated to receive a bit mask with 1 set for all applied FDs that have an outgoing edge as well. By intersecting with its negation, we get the value of the lowest bit that is set. If there is such a bit set, we use `LOG2MAP`, which is a map from $2^n \rightarrow (n+1)$, to lookup the Id of the lowest FD, which is then used to transition to the next DFSM state.

Visualization of NFSM and DFSM

The finite automata generated by the ordering manager quickly become large and complex and hard to read in a textual format. We have therefore, both to aid ourselves during development and for the future optimization rule developers, added functionality to visualize the automata in a graphical format. The automata are exported in the *dot* format, and we use Graphviz [AT 08] to turn them into graphs. The figures used in this chapter are all auto-generated.

7 Example Optimizations

"Science is what we understand well enough to explain to a computer. Art is everything else we do."

— Donald Knuth

To get a better understanding of how all the pieces of the optimizer work together, we first provide a fairly complete walkthrough of how a realistic query flows through the optimizer. Then, we provide a few sample queries that demonstrate the optimizer's abilities.

In this chapter we will not distinguish between "query" and "multi-query" — "query" refers to a MARS query graph which might have multiple outputs.

7.1 Walkthrough Query

Figure 7.1 shows an example of a realistic input query. It searches for documents containing the terms *"microsoft"* and *"software"*. Results are returned ordered by relevancy, as defined by the calculations of ScoreOccurrences, with the name of the documents. Also, the numbers of hits per document type are returned. We have removed the operator details to have it fit on one page.

This is the query we consider for the most of this chapter. Other queries are optimized in the last section. As described in Appendix C, we have included the debug output of the optimization of this query in the digital appendix. However, do not expect everything to be identical, as rule Ids and attribute names vary somewhat from run to run.

7.2 Optimization Steps

7.2.1 Optimization Hook

As described in Section 3.10, the optimizer is invoked through an OptimizerFacade, which handles converting to- and from our and MARS's graph structures. It first does some sanity checks, checking if the structure of the query is what we expect.

The query is well-defined and valid, so then, the query optimizer is invoked. The result of the optimization is converted back to a MARS operator graph before yielding processing to the next component in the chain.

7.2.2 Optimizer Initialization

Before optimization can be performed, the optimizer must be initialized. It is first instantiated by the facade. First, all rule binders in available assemblies are found, initialized and mapped.

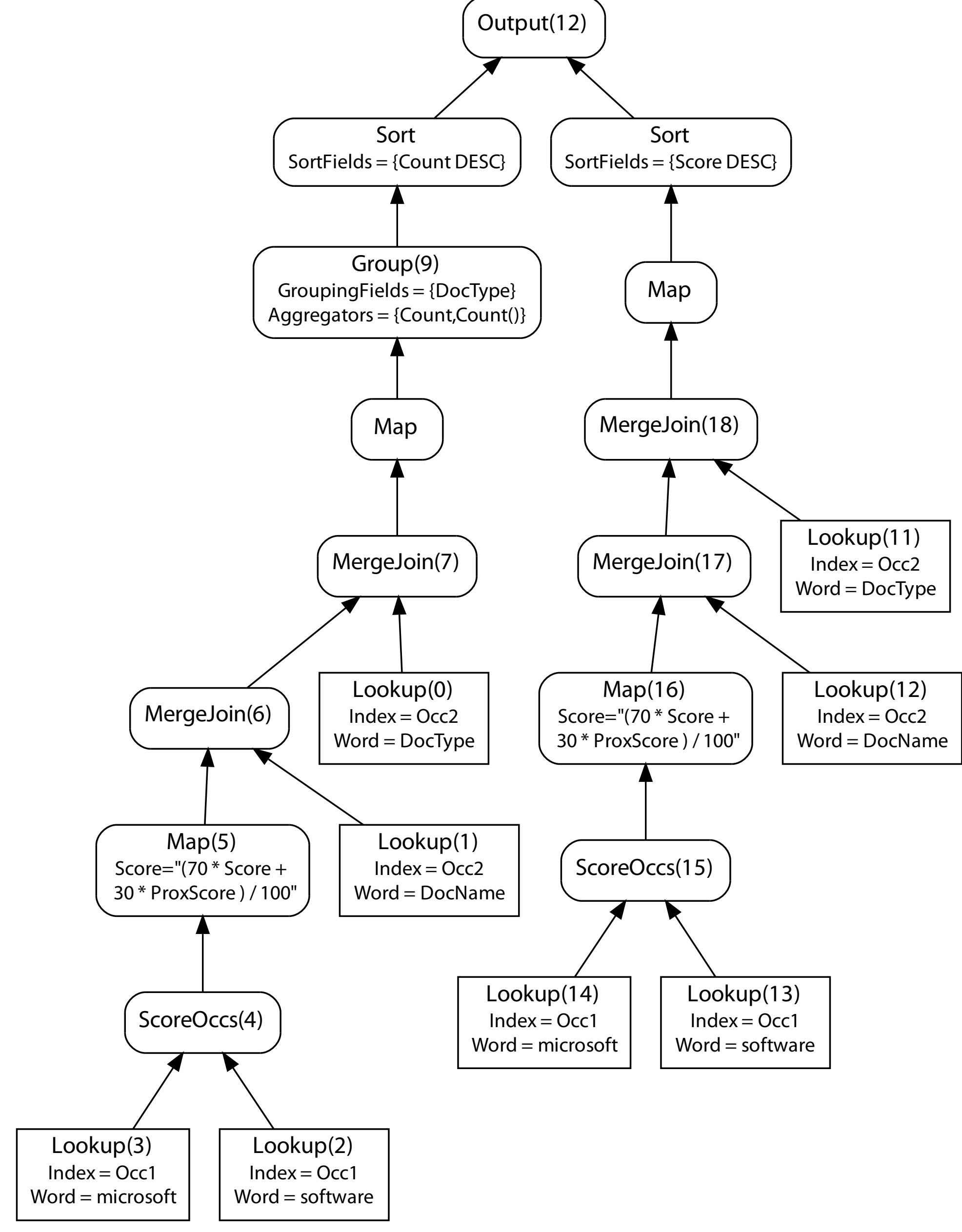


Figure 7.1: Example Query

Then, pre- and post-processors are found and configured. Since the processors can depend on each other, a topologically ordered processing chain is created. Note that reflecting for classes through available assemblies only happens on the first startup — not for every query.

7.2.3 Pre-processing

How the processors work are described in Section 3.5. Here we describe how they are applied to the example query.

At this point, the processing chain has the processors in the following ordering:

1. BehaviourMapperAnnotater annotates each operator whether it changes the result set, the record structure or the ordering.

2. CopyRemover does nothing in this case, but if a Copy-operator had been present, the query-DAG would have been converted into an equivalent tree.

3. LogicalJoinTransformer changes the MergeJoin-operators into logical Join-operators, to open for the opportunity that other join implementations are better.

4. ProducesAnnotator maps what attributes the operators produce, and if an operator passes an attribute through, which operator it originates from. Consider the $Score$-attribute. One attribute named $Score$ is produced by the ScoreOccurrences-operator, and another one is produced by the MapOperator above it. The $Score$-attribute referenced by the SortOperator is that of the MapOperator.

5. OrderingExtractor notes that the Output-operator requires its first child to be ordered on $Count$, which comes from the Group-operator, and that the other child must be ordered on $Score$. We need to map this, because Sort-operators are subsequently removed — they are treated as properties and not operators in the search phase. Only orderings that are guaranteed to make it to the output operator are kept. In this case, one of the outputs must be sorted on Count and the other on Score. An example of an ordering that would not have been conserved is a sort below a merge join. It is possible that a hash-join is optimal, so that ordering is just necessary to satisfy the requirements of the merge join operator.

6. SortRemover then removes the two Sort-operators. With the ordering requirements set on the other operators, the sort operators are no longer needed.

7. EquivalenceTransformer notes that the $DocumentId$-attributes of the Lookups are all equivalent, because they are equi-joined and/or scored. There are two equivalence classes, one for each branch.The information it produces is not of much use currently, but it is important to get the functionality explained in Section 8.5.5, about equivalence class joins.

8. DependencyAnnotator assigns operator dependencies transitively. For the sub-graph below the MapOperator defining Score, the operators depend directly on their inputs. However, consider the GroupOperator that groups on DocType. It depends on neither the join or the map above the join. It does, however, depend on the upper Lookup, since that is the operator which produces the attribute (and thereby also the join, since the join is the only way to add the Lookup to the rest). Disregarding renames, nothing but the Output-Operator actually depend on DocumentName. Noting this is important to be able to get to the goal shown in Figure 7.6.

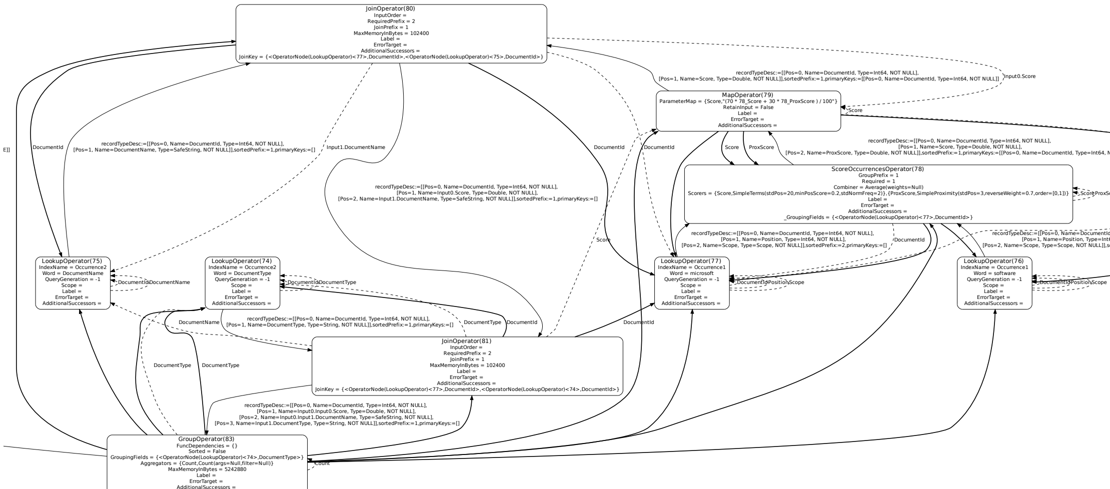


Figure 7.2: Illustration of the *amount* of information of node relations (not intended to be readable)

The output of the pre-processing phase is an operator graph that has the necessary properties and information for the rule binders to instantiate rules. Figure 7.2 shows an overview of the relations for the Group-operator and its descendants. We included it to show the degree of the relations between the operators, and obviously not because it is informative. The edges are information about dependencies and attribute origins.

7.2.4 Plan Generation: Preparation Phase

The preparation phase takes the input from the pre-processing phase and prepares for plan generation. First out is rule instantiation and configuration.

For every rule binder the optimizer has found, the rule binder's patterns are matched against the query graph. The rules instantiated by the binders are listed in Table 7.1. The Lookup rules are *base rules* — i.e. leaf nodes, which have no requirements. The numbers in the requirement column indicates other rules which the operator depends on. For instance, rule #5, a map rule, depends on rules 2, 3, 4, 6 and 7. There is no point in even attempting to move the map below 2, 3 and 4, since it will not lead to a valid plan. 6 and 7 is there to lock it in place below the group, which requires the map's `Score` attribute.

At this point, the produces- and requires attributes are represented with (fairly) descriptive strings. Only a number, like 14, means "operator applied", while 14_DocumentId means attribute (column) DocumentId from operator 14.

Then, bit sets are minimized, For example, properties that are always produced together are merged into one bit property. For example, the properties *0*, *0_DocumentId* and *0_DocumentType* are merged into one property "*0*". This reduces the number of properties from 43 to 17. Afterwards, the attributes are converted into a more compact bit-set-representation, on which operations are executed more efficiently.

Next, share equivalence classes are constructed using the previously described algorithms. Table 7.2 lists the 8 equivalence classes constructed for the query. Since the two lower parts of this query performs exactly the same operations twice, we get equivalence classes for operators in these parts, where each class has one member from each part.

At this point, the interesting orderings and groupings are known, so the OrderManager is told to construct its state machine. It is too complex to explain, but we include it in Figure 7.3 for completeness and to illustrate the complexity.

#	Rule	Produces
0	Lookup(Occurrence2/DocumentType)	0, 0_DocumentId, 0_DocumentType
1	Lookup(Occurrence2/DocumentName)	1, 1_DocumentId, 1_DocumentName
2	Lookup(Occurrence1/software)	2, 2_DocumentId, 2_Position, 2_Scope
3	Lookup(Occurrence1/microsoft)	3, 3_DocumentId, 3_Position, 3_Scope
11	Lookup(Occurrence2/DocumentType)	11, 11_DocumentId, 11_DocumentType
12	Lookup(Occurrence2/DocumentName)	12, 12_DocumentId, 12_DocumentName
13	Lookup(Occurrence1/software)	13, 13_DocumentId, 13_Position, 13_Scope
14	Lookup(Occurrence1/microsoft)	14, 14_DocumentId, 14_Position, 14_Scope

(a) Base Rules

#	Rule	Produces	Requires
4	ScoreOccurrences	4, 4_Score, 4_ProxScore	2, 3
5	Map	5, 5_Score	4_Score, 4_ProxScore, 2, 3, 4, 6, 7
6	Join	6	Left: 3_Document_id, Right: 1_DocumentId
7	Join	7	Left: 3_Document_id, Right: 0_DocumentId
9	GroupOperator	9	0_DocumentType, 0, 1, 2, 3, 4, 5, 6, 7
15	ScoreOccurrences	15, 15_Score, 15_ProxScore	13, 14
16	Map	16, 16_Score	15_Score, 15_ProxScore, 13, 14, 15, 17, 18
17	Join	17	Left: 14_Document_id, Right: 12_DocumentId
18	Join	18	Left: 14_Document_id, Right: 11_DocumentId

(b) Search Rules

Table 7.1: Instantiated rules, their produced- and required attributes

Representative	Members
0 Lookup(Occurrence2/DocumentType)	0 Lookup(Occurrence2/DocumentType)
	11 Lookup(Occurrence2/DocumentType)
1 Lookup(Occurrence2/DocumentName)	1 Lookup(Occurrence2/DocumentName)
	12 Lookup(Occurrence2/DocumentName)
2 Lookup(Occurrence1/software)	2 Lookup(Occurrence1/software)
	13 Lookup(Occurrence1/software)
3 Lookup(Occurrence1/microsoft)	3 Lookup(Occurrence1/microsoft)
	14 Lookup(Occurrence1/microsoft)
4 ScoreOccurrences	4 ScoreOccurrences
	15 ScoreOccurrences
5 Map	5 Map
	16 Map
6 Join	6 Join
	17 Join
7 Join	7 Join
	18 Join

Table 7.2: Share equivalence classes

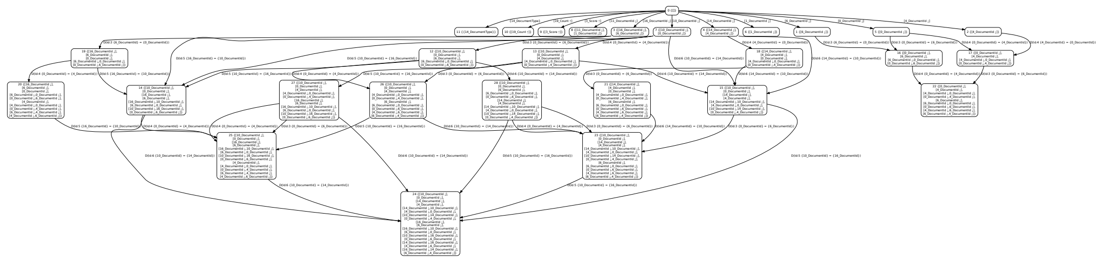


Figure 7.3: Orderings and groupings DFSM for the example query (not intended to be readable)

Query	Goal	Order
0	0, 0_DocumentId, 0_DocumentType, 1, 1_DocumentId, 1_DocumentName, 2, 2_DocumentId, 2_Position, 2_Scope, 3, 3_DocumentId, 3_Position, 3_Scope, 4, 4_Score, 4_ProxScore, 5, 5_Score, 6, 7, 9	[9_Count DESC]
1	11, 11_DocumentId, 11_DocumentType, 12, 12_DocumentId, 12_DocumentName, 13, 13_DocumentId, 13_Position, 13_Scope, 14, 14_DocumentId, 14_Position, 14_Scope, 15, 15_Score, 15_ProxScore, 16, 16_Score, 17, 18	[16_Score DESC]

Table 7.3: Query goals

The **goals** of the query are now determined as the union of the produced properties of all logical operators (which happens to be all of them in this case). Since this is a multi-query, we have two goals as shown in Table 7.3

Finally, before the search is started the memoization table is initialized and the base plans entered into the table. For each base rule, one base plan is inserted with the rule's *produced* bit set as key. To enable base plan sharing, the base plans for equivalence class representatives are inserted for both its own properties and for all the plans they represent as shown in Table 7.4 (we have only included four of the rules for brevity).

7.2.5 Plan Generation: Search Phase

For the given query, 41 rule appliances are used to generate 37 plans for 23 different goals (the actual number is much higher, since pruned, dominated plans are not counted), resulting in a debug output of 224 KiB. Explaining the whole search process here is therefore unfeasible.

Instead, we have chosen one example which illustrates how the plan generator and cost

Memoization Key (properties)	Plans
0, 0_DocumentId, 0_DocumentType	0 Lookup(Occurrence2/DocumentType)
1, 1_DocumentId, 1_DocumentName	1 Lookup(Occurrence2/DocumentName)
11, 11_DocumentId, 11_DocumentType	0 Lookup(Occurrence2/DocumentType)
	11 Lookup(Occurrence2/DocumentType)
12, 12_DocumentId, 12_DocumentType	1 Lookup(Occurrence2/DocumentName)
	12 Lookup(Occurrence2/DocumentName)
...	

Table 7.4: Base plans

model works together. Use Figure 7.1 (page 110) to follow this example. Below, we have included an excerpt from the debug output where join rule 18 is searching for its left input goal, namely [12, 13, 14, 15, 17], by asking the plan generator to produce plans for satisfying it. The plan generator answers by first rewriting it to [1, 2, 3, 4, 6] using equivalence class representatives. This yields a cache hit, as this plan has been produced before. The result is a PLAN-ADD for plan-11 in the PlanSet (11 is just a plan counter, not related to the goals), as this is the first plan found. It is actually MergeJoin(6) from the other part of the query, offering to share the entire sub-plan, including itself (6 is in the *sharing* bitset, and *shared* is true).

Next, join rule 17 is invoked to produce plans by the plan generator. It answers by asking the plan generator for [13, 14, 15] on its left input, which yields a cache hit. Then it asks the plan generator for [12] for its right input, which also yields a cache hit (this is a base plan and will always be in cache).

Join rule 17 has now both its inputs satisfied and proceeds by constructing two physical plan alternatives for the same logical goal: MergeJoin(17) and HybridHash(17), which it adds to the PlanSet. Since the inputs are ordered in this case, we can see that the merge join is cheapest. On the last two lines we can se that the PlanSet actually rejects both plans, effectively pruning them away immediately. They are both dominated by Plan-11 which has lower or equal cost and offers more sharing (we have left out ordering in the excerpt for brevity). This is because Plan-11 was constructed with the class representative Join(6), and the others were not. Note that Plan-61 has the same cost as Plan-11. This is because it is the same plan, but with Join(17) on the top instead of Join(6). Join(18) has now got one plan for its left input and the search proceeds by considering other rules than Join(17) for application.

```
JoinRule producing [18] searching for [12, 13, 14, 15, 17] LEFT.
  Rewrote [12, 13, 14, 15, 17] to [1, 2, 3, 4, 6], firing off share equivalent search.
    CACHE return: [1, 2, 3, 4, 6]
   ADD: Plan-11(MergeJoin(6), Cost: 3925, Sharing: [1, 2, 3, 4, 6], Shared: True)

   JoinRule producing [17] searching for [13, 14, 15] LEFT.
     CACHE return: [13, 14, 15]
   JoinRule producing [17] searching for [12] RIGHT.
     CACHE return: [12]

   JoinRule producing [17] adding plans...
  NO-ADD: Plan-61(MergeJoin(17), Cost:3925, Sharing:[1,2,3,4], Shared:False)<<Plan-
11
  NO-ADD: Plan-62(HybridHash(17), Cost:4000, Sharing:[1,2,3,4], Shared:False)<<Plan-
11
```

7.2.6 Plan Generation: Reconstruction Phase

When the optimizer is done generating plans, the final plan is represented as a Plan, which is a light weight structure. Before post-processing can proceed, we must convert the plan structure into an OperatorNode-graph.

This is done by recursively calling *BuildAlgebra* on the rules. The rule binders have made sure that sufficient information to reconstruct the operator node is stored in the rule, so the information from the Plan-structure is just what its inputs plans are.

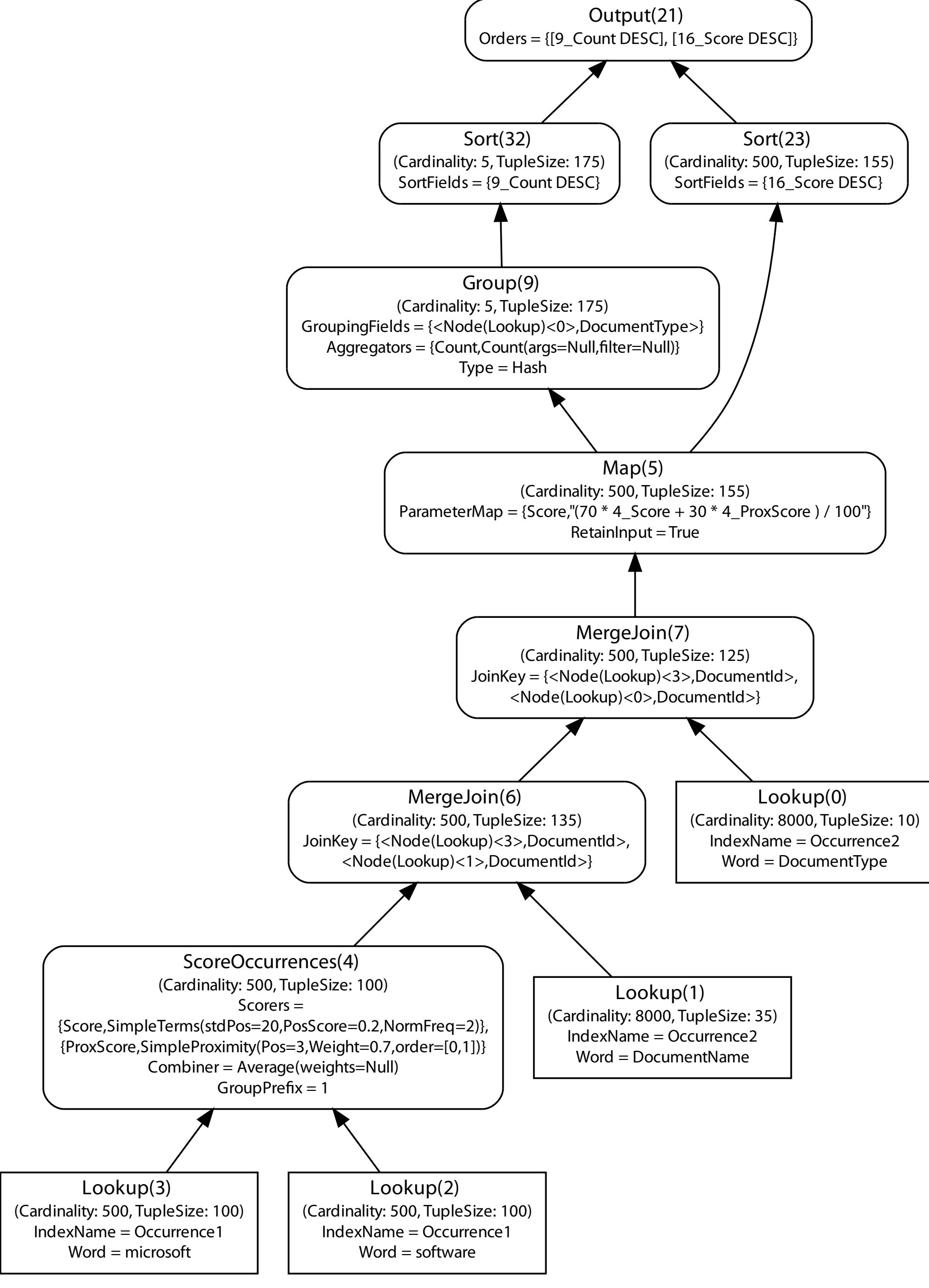


Figure 7.4: Reconstructed query, due for post-processing

7.2.7 Post-processing

Figure 7.4 shows the output of the reconstruction phase — a query graph with physical operators. A lot of information has been omitted to make it fit on one page.

Before the query can be executed, we need to update attribute labels and add a Copy-operator above any operator with multiple outputs. This amounts to adding a Copy-operator above the Map-operator with two outputs, and adding a Map-operator above each child of the Output-operator. The Map-operators added ensure that the attribute names in the final query equal those of the input query, and that any removed attributes do not appear. This is necessary since search rules are not instantiated for Map-operators that only remove and/or rename attributes, and because we need to do our own (node, attribute)-mapping of attributes when constructing entirely different graphs from the input.

The result of the post-processing, and consequently the entire optimization, is shown in Figure 7.5.

7.2.8 Returning to MARS

As mentioned briefly in Section 7.2.1, we must convert our query graph structure to the one MARS uses. This is done by the optimizer facade, and not by the optimizer itself. The optimizer is oblivious to any other graph structures but its own.

When the graph is converted, the OptimizerTransformer-component added to MARS' query pipeline replaces the query to be executed with the optimized one.

7.3 The Effect of the Optimization

Now we want to focus on how the optimizer actually improves query evaluation time and show how useful and efficient DAG-structured query plans really are.

To benchmark this, we use largely the same query, but we limit the amount of document results to 10 and vary the lookup terms. The trim to 10 results is not visible before Figure 7.6 since we wanted to keep the previous example simpler. To recap, the query does the following:

Sub query 1 Search for all documents containing *word1* and *word2*, scoring each document and returning the top 10 documents with the best score.

Sub query 2 Search for all documents containing *word1* and *word2*, grouping them by document type and returning the number of documents of each type.

7.3.1 Query versions

We now continue by benchmarking the query evaluation time for three logically equivalent, but physically different versions of this query: 1) Completely unoptimized, tree-structured query, 2) suboptimal DAG-ified query and 3) optimal DAG-ified query.

Figure 7.1 shows the query as it is input to the optimizer. Large portions of the graph perform exactly the same work. They are share equivalent and can be merged, so here we have good opportunities to perform better.

Figure 7.5 shows the same query, but now with the share equivalent parts of the query shared using the copy operator. In addition, the optimizer has considered hash joins and hash group, but those are clearly less efficient since the lookup operators provide sorted data.

Figure 7.6 shows one further optimization that can be performed to the query in Figure 7.5, i.e. moving the join with the DocumentName lookup up and above the sort operator which

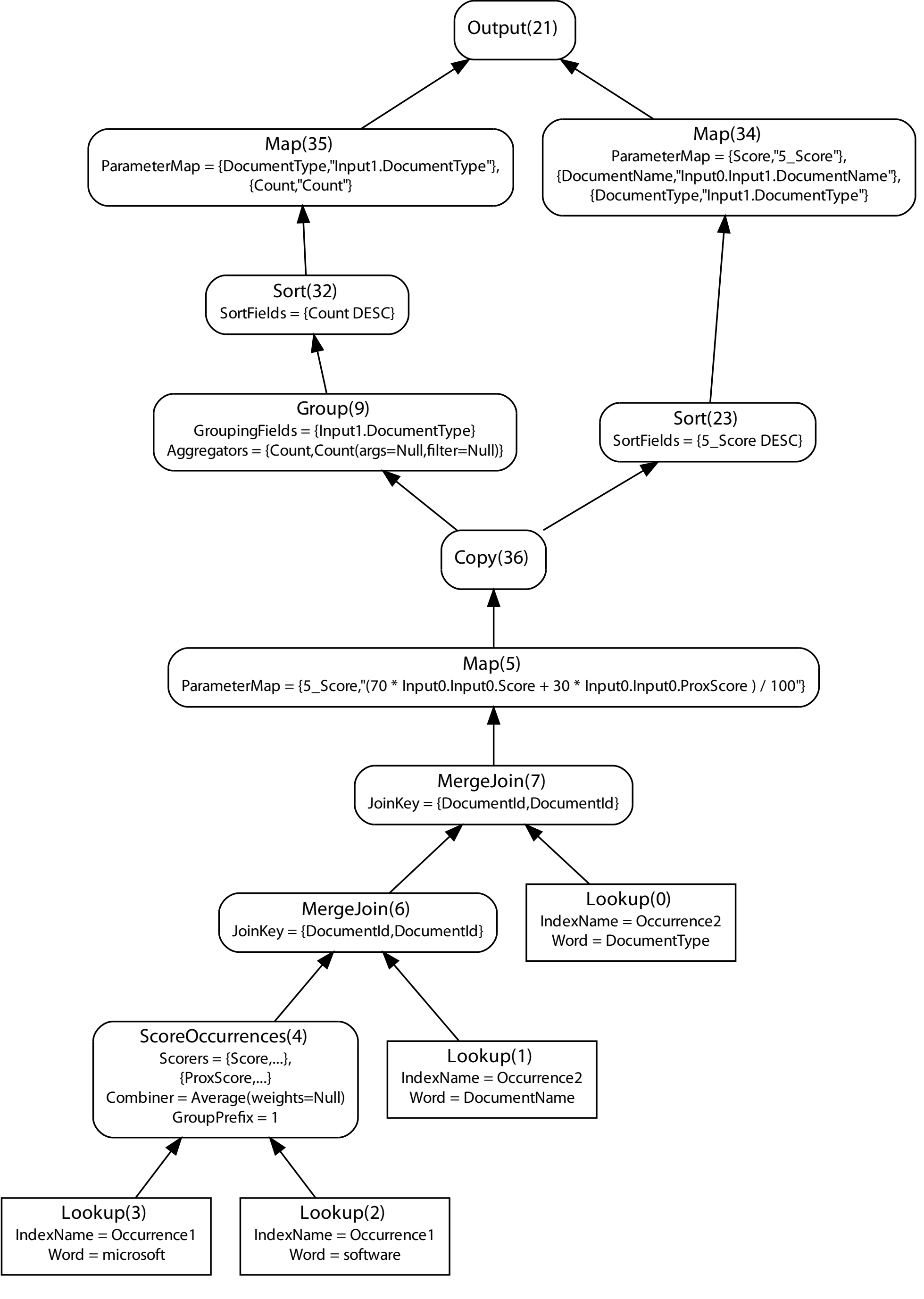


Figure 7.5: Final query graph, due for execution

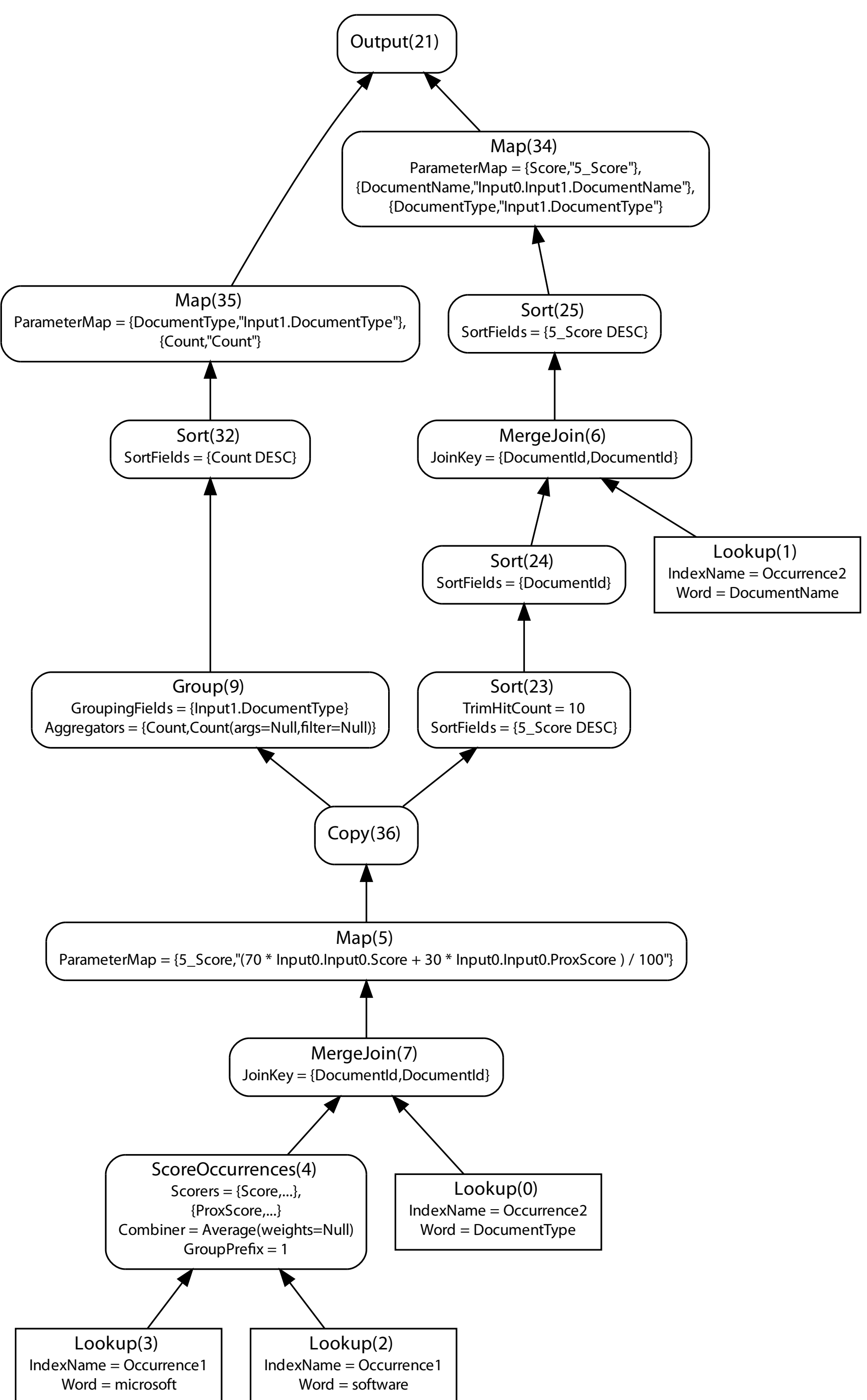


Figure 7.6: Optimal DAG-ified query

also happens to trim the record set to only 10 tuples. This means that the join will be less expensive, since it is now only working on 10 tuples instead of possibly several thousand.

However, we consider this to be an advanced optimization, since performing it requires knowledge about foreign key relationships. The optimizer has to be sure that the join with DocumentName does not alter the number of tuples, since it would otherwise yield different outputs for the query if it is moved through the logical trim operator that is embedded in the sort operator. We include benchmarks of it for completeness, though. How we envision it implemented is described in Section 8.5.6.

7.3.2 Benchmark setup

All queries were run against the same full-text index. To get realistic results, we have indexed a subset of the articles found in Wikipedia [Wik09]. The articles have been pre-processed before indexing to extract meta information from the article text, like identifying person names, places, and so on. The resulting information is stored as XML files on disk before being read by the indexer. An example is shown in listing 7.1

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <documents xmlns:ann="http://annotations" ann:mode="InlineWithOverlaps">
3    <document>
4      <documentId>doc038652.xml</documentId>
5      <documentType>sa</documentType>
6      <title><paragraph><sentence>Pelee</sentence></paragraph></title>
7      <body>
8        <paragraph><sentence>Pelee</sentence></paragraph>
9        <paragraph><sentence><![CDATA[Originalwikipedia article => "Pelee"]]></sentence></paragraph
           >
10       <paragraph>
11         <sentence>A volcano located in West Indies.</sentence>
12         <sentence>It is famous for its 1902 eruption which took lives of 29,000 people.</sentence>
13       </paragraph>
14       <paragraph><sentence>See Mt.Pele</sentence></paragraph>
15     </body>
16   </document>
17 </documents>
```

Listing 7.1: Pre-processed Wikipedia article.

The index contains approximately 250,000 documents, which has a raw XML size on disk of 1.57 GiB. The reverse index on word occurrences totals 1.19 GiB, while the size of the document type and document name indexes are 29 MiB.

The test system is a Intel Core Duo T2500 (2×2.00 GHz) with 3 GiB DDR2 memory, of which approximately 1 GiB was allocated to MARS when running the benchmarks. The data was stored on an 80 GiB Intel X25-M Solid State Drive.

The reverse index lookups themselves (the Lookup operators at the bottom) are fairly cheap due to the index structure as long as the number of results (i.e. the number of hits in each document summarized) is fairly small. The same is true for the evaluation of the query graph — as long as the number of tuples flowing through it is relatively small, the query evaluation costs are fairly small. To get results that are reliable and not adversely affected by setup time and variance, we therefore need to query for search terms that return a larger amount of hits.

However, if the number of hits is too great, we may hit the memory limits on the test system and have the operators spool data to disk (especially the Copy operator), creating results that are hard to analyze. We therefore decided to plot query evaluation time against the expected number of hits for the search terms used.

Query No	Word 1	Word 2	Est. hits
1	gaza	1978	9 600
2	1978	today	19 212
3	today	charles	38 425
4	charles	about	77 025
5	about	their	156 925
6	their	have	312 212
7	have	by	673 512
8	by	is	1 550 112

Table 7.5: Benchmarking queries

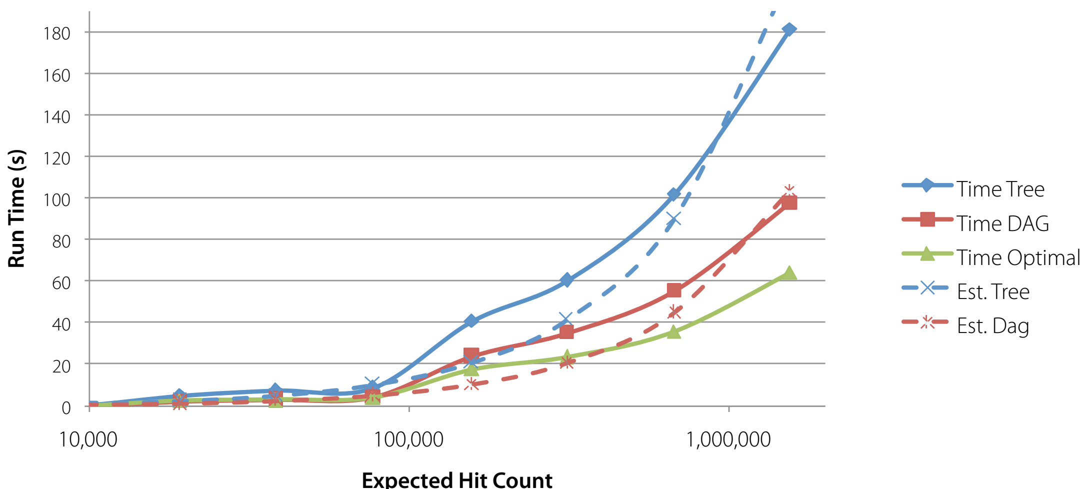


Figure 7.7: Benchmark results

We approached this by counting the word frequencies in a subset of 20,000 documents, thereby estimating the number of hits for each word to be $250,000 \times \frac{f}{20,000}$. The estimated number of index word hits (not output result count) for a query for *word1* AND *word2* then becomes $250,000 \times (\frac{f_1}{20,000} + \frac{f_2}{20,000})$. We then selected a random word at or near word frequencies of 2^n hits (up to the maximum word frequency) and ran each of the three physical queries for these two words, averaging the evaluation time over several runs. We measured standard deviations and found them to be insignificant

We realize that assuming independence of the terms is sub-optimal, as explained in Section 4.3. However, we had to settle on something simple. Table 7.5 lists the different queries run and their expected hit counts.

7.3.3 Results

An overview of the results from the benchmark can be seen in Figure 7.7. We have excluded very small expected hit counts from the plot, as they are very sensitive to noise. The main point that can be seen from the plot is that the *Query A - Tree* is more expensive than *Query B - DAG*, which again is slightly more expensive than *Query C - Optimal DAG*. This is approximately what we would expect and confirms that DAGs are worth pursuing. If we look closer, we can actually see that query A is overall around twice as expensive as the other two queries, which makes sense, since query A performs the most expensive part of the query twice instead of once.

We have also included the costs predicted by our cost model (dashed lines) for *A - Tree* and

	Timings in milliseconds			Estimated costs in ms		% Off	
Query	Tree	DAG	Optimal DAG	Tree	DAG	Tree	DAG
1	257	124	165	1 640	820	538	561
2	5 378	2 786	2 758	2 924	1 462	-46	-48
3	7 959	3 687	3 380	5 489	2 744	-31	-26
4	9 149	4 853	4 230	10 642	5 321	16	10
5	41 051	24 322	17 678	21 310	10 655	-48	-56
6	60 675	35 489	23 715	42 042	21 021	-31	-41
7	101 951	55 596	35 945	90 280	45 140	-11	-19
8	181 140	97 824	64 234	207 317	103 658	14	6

Table 7.6: Detailed results

B - DAG. We have no estimates for *C - Optimal DAG*, as we have not implemented support for this optimization yet. We can see that the predicted costs follow a more predictable path than the actual timings, but they are not too far off. However, it is not the absolute numbers that is most interesting in this context, it is rather the relationships between the costs of different plans. The optimizer does not care about the absolute costs, it only cares about which plan is cheapest. We do not say absolute values are irrelevant — the more accurate, the better — but there are other things that are more important than entirely accurate guesstimates.

With this in mind, by looking at the dashed lines, we can conclude that the optimizer has successfully identified the *B - DAG* plan to be approximately half the cost of the *A - Tree* plan, which conforms well to the reality.

Table 7.6 shows the detailed timings along with the costs our cost model predicts for each of the queries for the different hit counts. The last column shows the deviation of actual to estimated timings in per cent of the actual timings. We can see that the deviation is not too great and mostly within +/- 50%. The exception is the first query, but here the time is so short (well below half a second), so it is very prone to noise and any buffering the operating system may have done.

7.4 Additional Optimizations

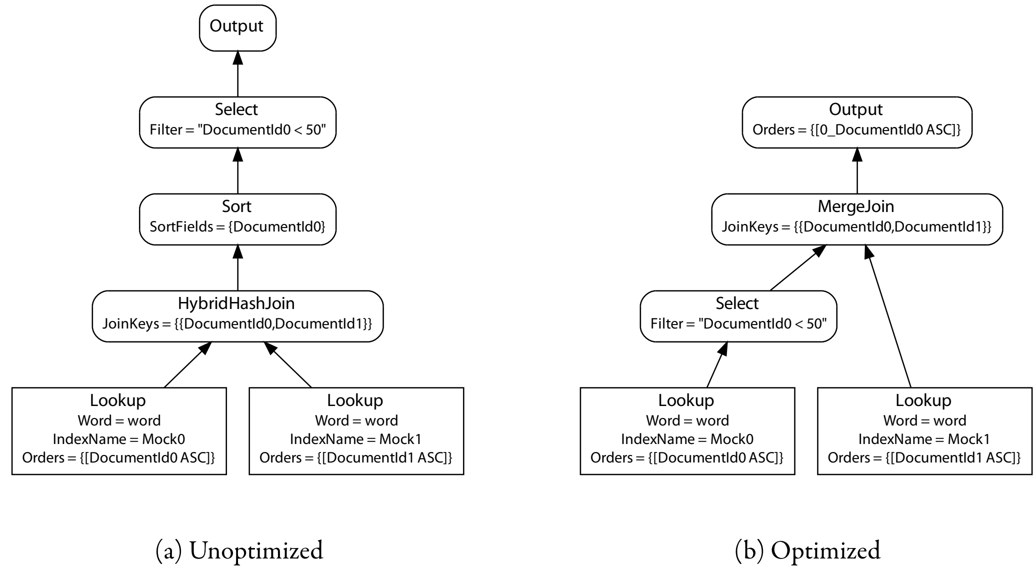
To display the variety of optimizations the optimizer can do, we include a few more examples. Note that the example queries are selected by their value as examples, not necessarily because they are realistic real-word queries. Real-word queries rarely demonstrate as many concepts at once, and usually require more context to understand.

7.4.1 To Sort Or Not To Sort

Figures 7.8–7.10 show a few queries where existing orderings or orderings induced by functional dependencies affect whether a sort is performed and what join algorithm is used.

Consider Figure 7.8. In this query, the indexes are clustered on their "DocumentId"-attributes. By replacing the hash join with a merge join, this order is kept also through the join, so the Sort-operator can be removed entirely, in addition to the join itself being cheaper.

Figure 7.9 shows a query where only index "Mock1" is clustered. The lookup on "Mock0" has no physical ordering. The optimizer finds that sorting the unsorted input to the join and switching to a merge join is the cheapest. Note also how the select operator has been pushed down.



(a) Unoptimized (b) Optimized

Figure 7.8: Query where the sort is removed because of the existing orderings

In the last case, there are no existing orderings, but it is inferred by the predicate on the select (`DocumentId1` = 50, leading to → `DocumentId1`) and the join predicate (`DocumentId0` = `DocumentId1`, leading to → `DocumentId0` as well), so the Sort-operator can be removed.

We did some quick performance testing with regards to removing redundant sort operators by executing a query on the word *is*. The before-query was `SORT("Position ASC";-SELECT("Position=30"; LOOKUP()))`, while the after query `SELECT("Position=30"; LOOKUP())` since the sort was really redundant. This yielded a performance improvement from **4.90s** to **1.99s**, even though the selection was placed before the sort in the first query.

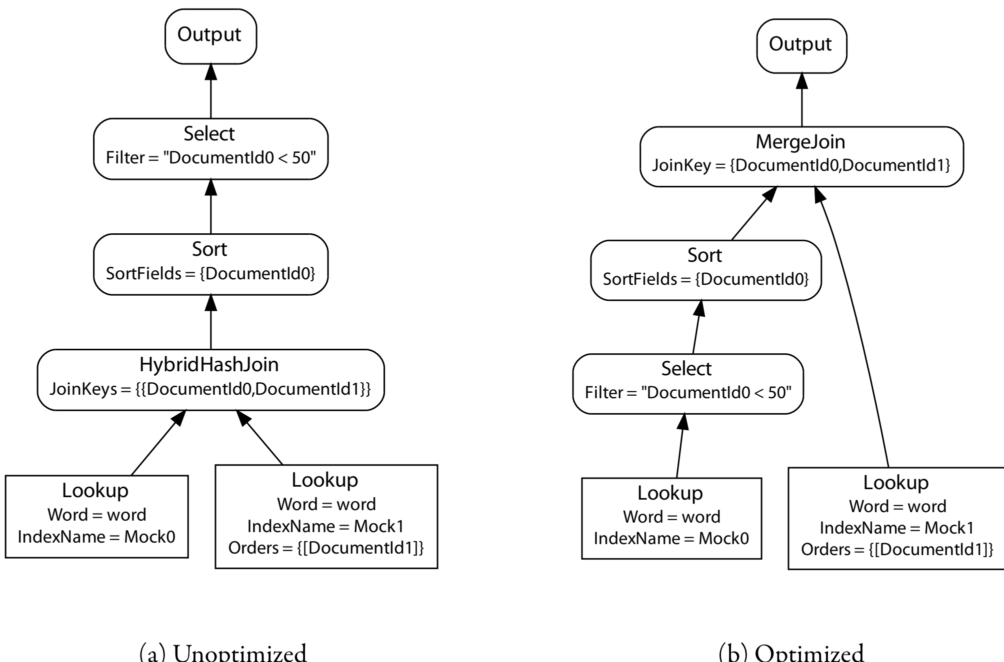
7.4.2 Join Ordering

One of the most important aspects of the optimizer is the capability of ordering joins to get the cheapest plan possible. In this example, we show how our optimizer figures out the optimal join order for a simple query with three joins. The query would roughly correspond to the SQL query `SELECT * FROM R0 JOIN R1 ON R0.Id0 = R1.Id1 JOIN R2 ON R1.Id1 = R2.Id2 JOIN R3 ON R2.Id2 = R3.Id3`. See Section B.6 for the automated optimizer test that was used to generate the following figures.

The output cardinality for a join $\mathtt{A} \bowtie \mathtt{B}$ is given by $|\mathtt{A}| \times |\mathtt{B}| \times \mathcal{S}(\mathtt{A}, \mathtt{B})$, where $\mathcal{S}(\mathtt{A}, \mathtt{B})$ is the *selectivity* of the join. The *cost* $\mathcal{C}(\mathtt{A} \bowtie \mathtt{B})$ for a join is given by $\mathcal{C}(\mathtt{A}) + \mathcal{C}(\mathtt{B}) + \alpha|\mathtt{A} \bowtie \mathtt{B}|$, that is, the sum of the children costs plus the cost of the join, where α is a factor that weighs the cost of the join itself.

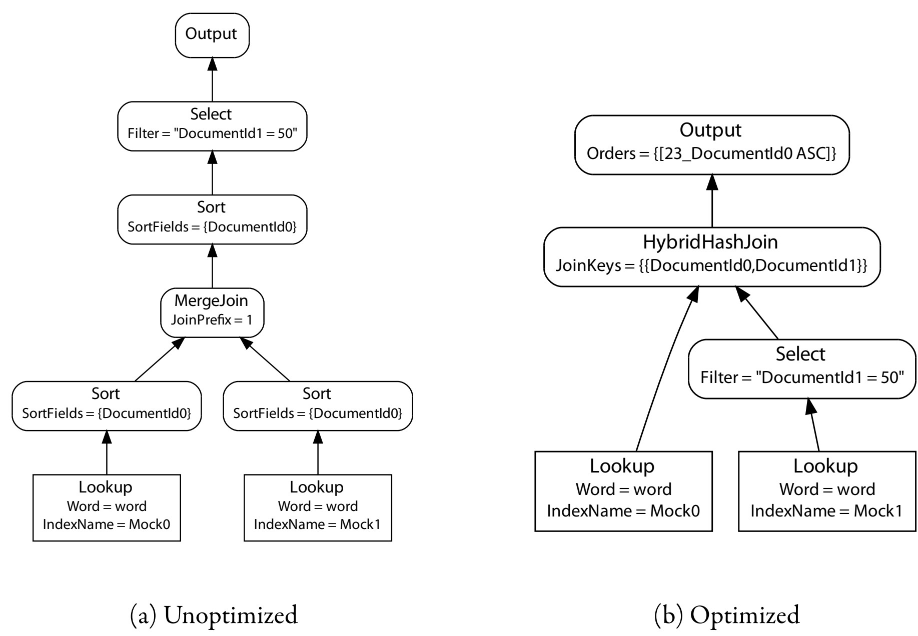
As input to the optimizer, we give it the worst case plan, $((\mathtt{R1} \bowtie \mathtt{R2}) \bowtie \mathtt{R0}) \bowtie \mathtt{R3}$, as shown in Figure 7.11. First, we tell the optimizer to only consider left-deep plans. The result can be seen in Figure 7.12. The optimizer has chosen to switch the order of the joins $\mathtt{R0} \bowtie \mathtt{R1}$ and $\mathtt{R1} \bowtie \mathtt{R2}$, as $\mathtt{R0} \bowtie \mathtt{R1}$ has much higher selectivity and thereby *limits the cardinality of the temporary result in between them, keeping the cost down.* The cardinality of the joins in the input query were 20000 at its largest, whereas it in the optimized query is reduced to 2000.

One may wonder why the other very selective join was not moved down. The reason is that this is a chain query where the result from the bottommost join is R0, R1. Neither R0 nor R1 can be joined directly with R3 (R3 must be joined with R2 first), so putting this join as the



(a) Unoptimized (b) Optimized

Figure 7.9: Sort is kept because there is no existing ordering



(a) Unoptimized (b) Optimized

Figure 7.10: Ordering derived from functional dependencies

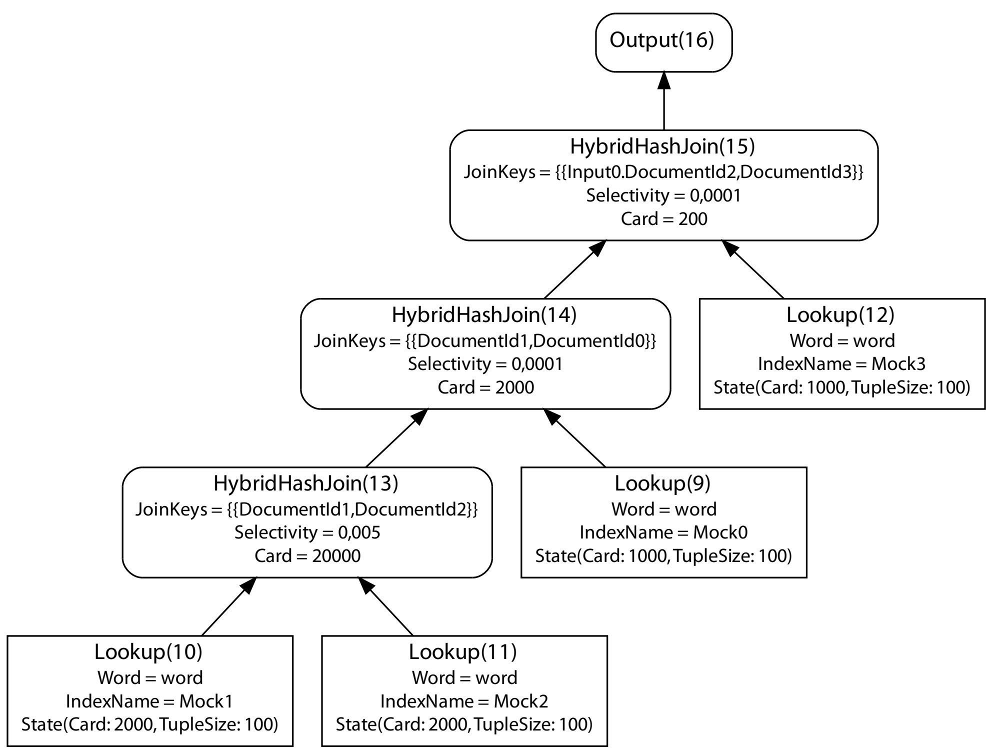


Figure 7.11: Query before join ordering optimization

second join from the bottom would give us a cross product. This is certainly not a good idea, as cardinality and thereby cost would increase.

Now, we tell the optimizer to search for any plan, including bushy plans. This dramatically increases the search space (see Section 1.6.1 for details), but it is not a problem for this small query. This time, the optimizer can do what it could not the last time. It now selects to do both of the selective joins first, then joining the result together to get the final result. This turns out to be even better. The largest cardinality for a join is now reduced to 200, as seen in Figure 7.13

As we can see, the expected output cardinality for the query is the same in all cases.

7.4.3 Multi-Query Optimization

As mentioned, optimizing multi-queries has been a high priority. This is a defining feature of MARS, and most design decisions were taken with regard to supporting it — already last year when we implemented a bare-bones optimizer.

Figure 7.14 shows the input to the optimizer. It is a combination of the two previous examples — i.e. the same join graphs with different orderings and a selection. There is also a select operator with low selectivity present.

Figure 7.15 shows the optimized version. Note how the select operator is now above the Copy-operator. By keeping a partial plan that is more expensive, but allows more sharing, we get the globally optimal query. If we had only kept the cheapest plans without regard to sharing, we would not have been able to devise this plan.

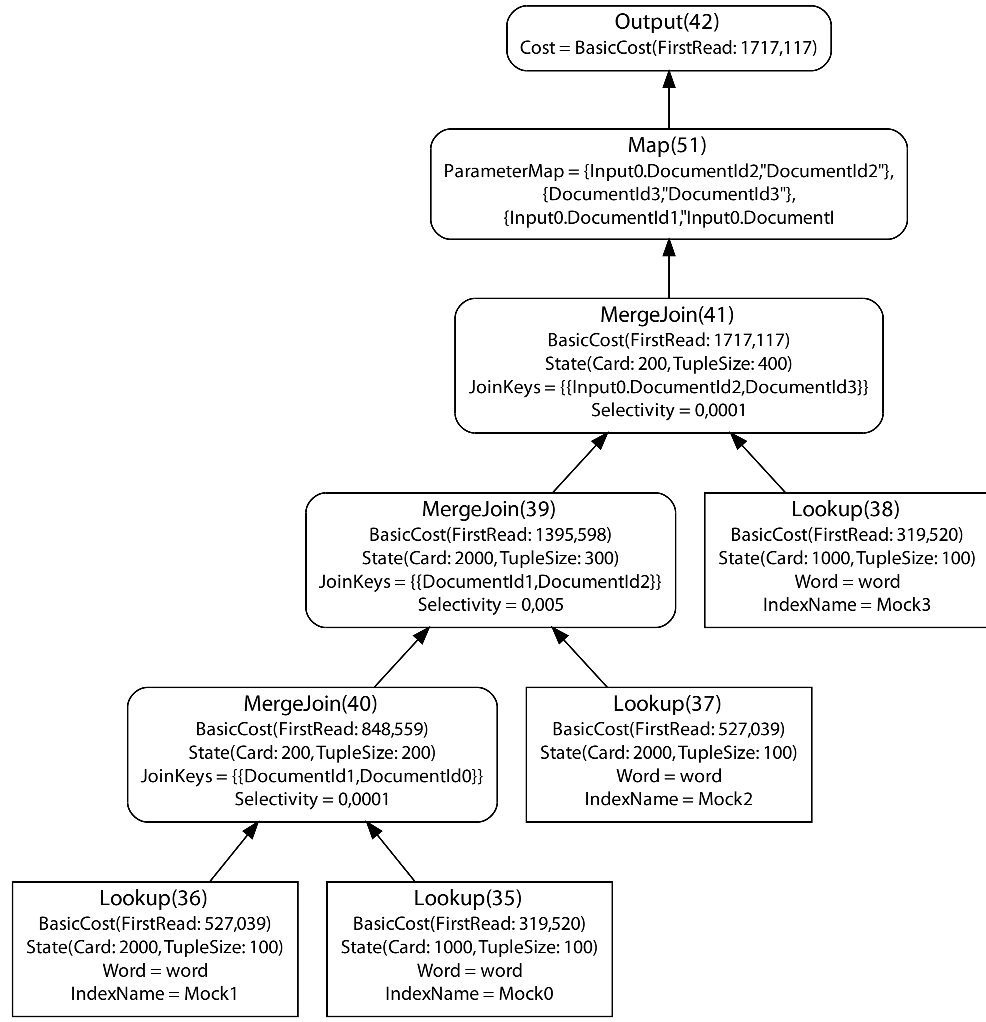


Figure 7.12: Query after optimization, only considering left-deep plans.

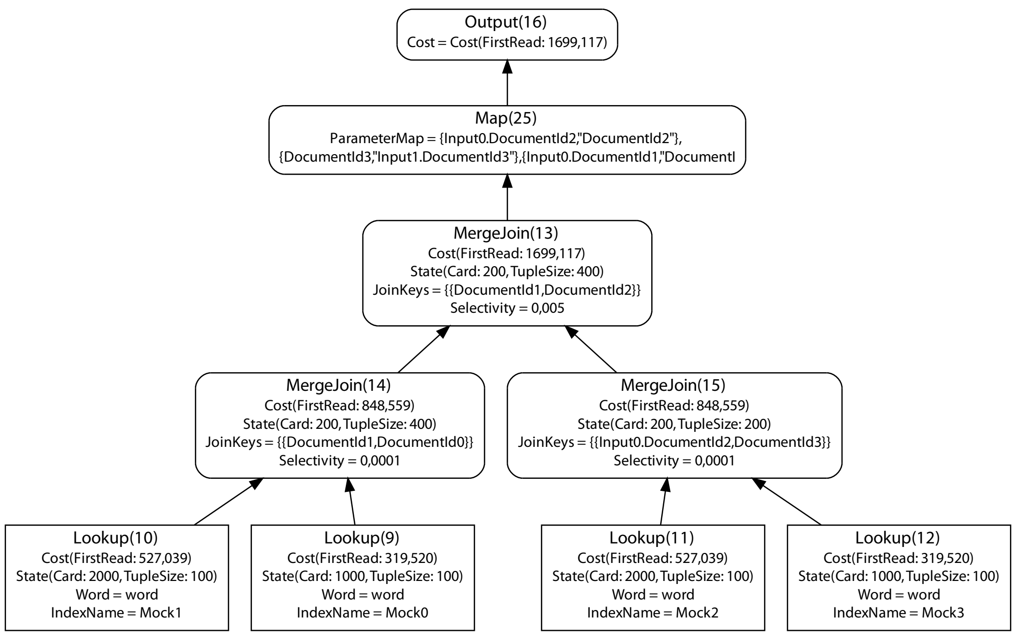


Figure 7.13: Query after optimization, also considering bushy plans

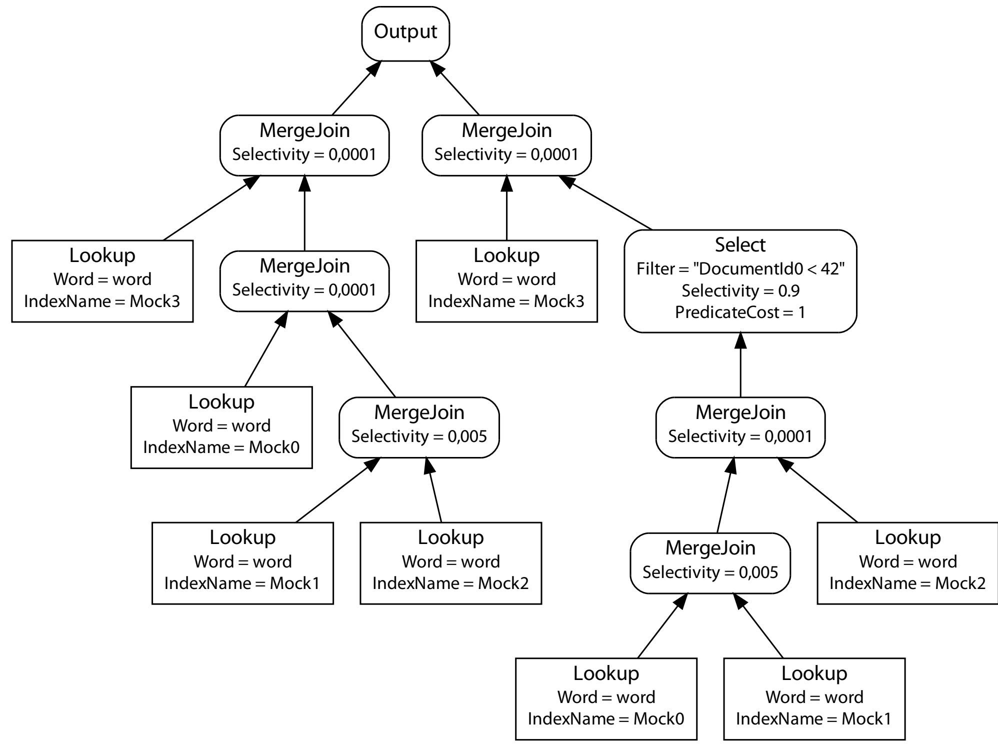


Figure 7.14: Input multi-query

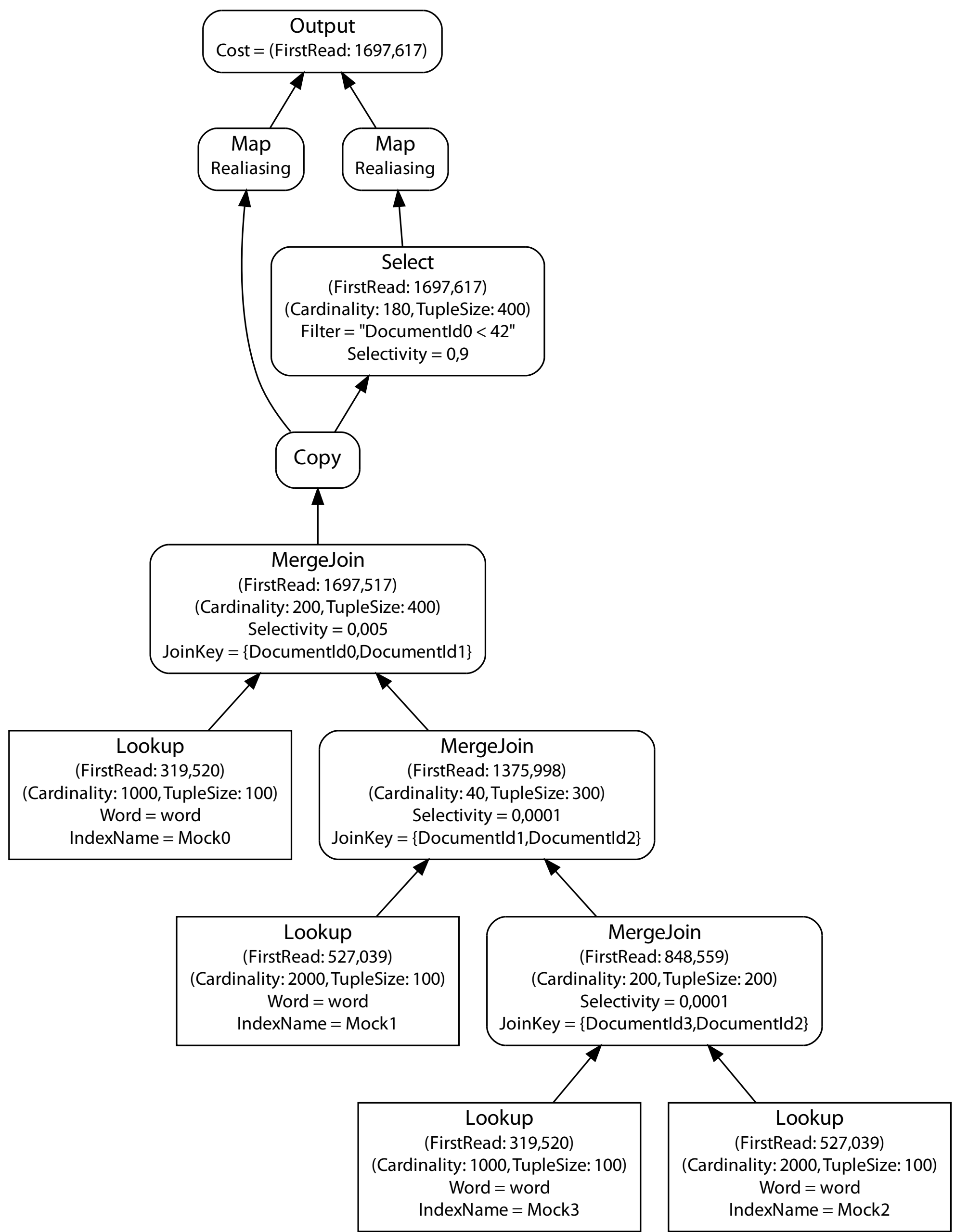
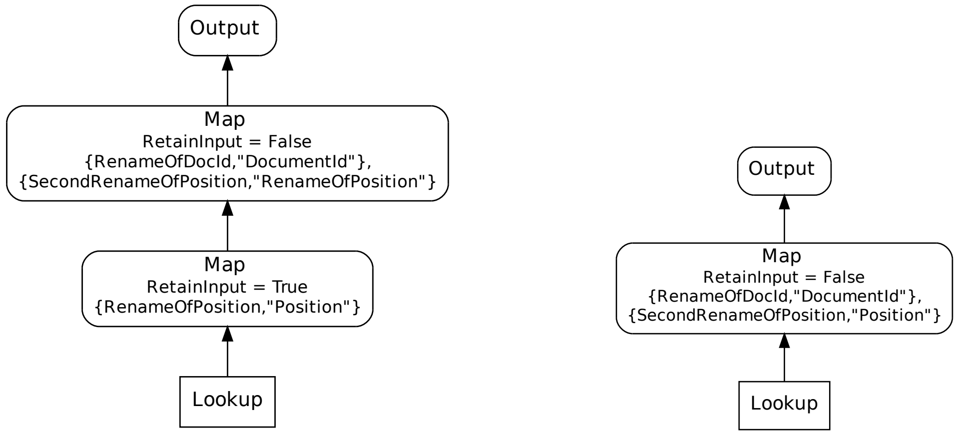


Figure 7.15: Optimized multi-query



(a) Input query with two map operators. (b) Optimized query, only on map operator.

Figure 7.16: Map merging

7.4.4 Map Merging

It is perfectly valid, but not very efficient to have multiple consecutive `Map` operators in a query, as these operators can be merged quite easily. Our optimizer is currently able to merge simple map operators which only limit (project) or rename attributes. Actually, this is not performed explicitly, it comes as a bonus from the way we handle attribute references. Remember that during pre-processing, all attribute references by name are replaced by `NodeAttribute` references, which refers to the attribute as `<Source node, name>`. The algorithms handling this replacement also handle attribute renames in map operators, even transitive ones (where an attribute is renamed multiple times). Other operators between the map operators do not disturb this algorithm. During plan generation, all simple map operators are removed and only the necessary ones are reinserted.

Figure 7.16a shows an input query with two consecutive map operators. The lower one renames the `Position` field, while the upper renames the `DocumentId` field, renames `Position` again, as well as projecting away all other attributes. In the optimized query in Figure 7.16b, the two map operators have been merged into one.

8 Current State

"It takes an awful long time to not write a book."

— Douglas Adams

In this chapter, we present the current state of our optimizer implementation. We start out by summarizing what we have achieved, before talking about the issues we have identified and proposed solutions to them.

8.1 Results

In short, we have implemented a proof-of-concept optimizer which deals with multi-queries and DAG-structured queries, cleverly handles orderings and groupings, and performs various join-orderings. It is also based on an architecture which addresses the design goals introduced in Section 3.1. It is *extensible* and the rule architecture enables support for arbitrary operators and pre- and post-processors, while the cost model is external to the optimizer itself. Its performance is acceptable, although not the best in its class, since this has not been the primary focus.

We have identified a few issues and have proposed solutions to them, as explained in Section 8.3. Performance is a challenge to all query optimizers because the general problem is exponential in nature.

8.2 Performance

Since the time spent on query optimization is included in the query response time, and thereby the final response time to the user of the application, it is important that the query optimizer is performing well. The problem can be remedied somewhat by caching query plans, but still, whenever a new query is to be run, it must be optimized. Response time is especially important in our case with MARS, as search engine users generally expect short response times.

At the same time, the problem of query optimization (including join enumeration) has exponential complexity with respect to the number of relations. Therefore, the optimizer needs to be aware of its own expected run time and be able to switch to alternative strategies which run more quickly, but may yield potentially suboptimal plans. Examples of such strategies are to only consider left-deep plans, or switch to non-exhaustive heuristics when the expected complexity is too great.

To get an overview of the performance of the optimizer, we have done some simple benchmarking. We construct a query to optimize by adding more and more relations to be joined. *Star queries* are queries where all relations $1..n$ are joined to relation 0, while in *chain queries*,

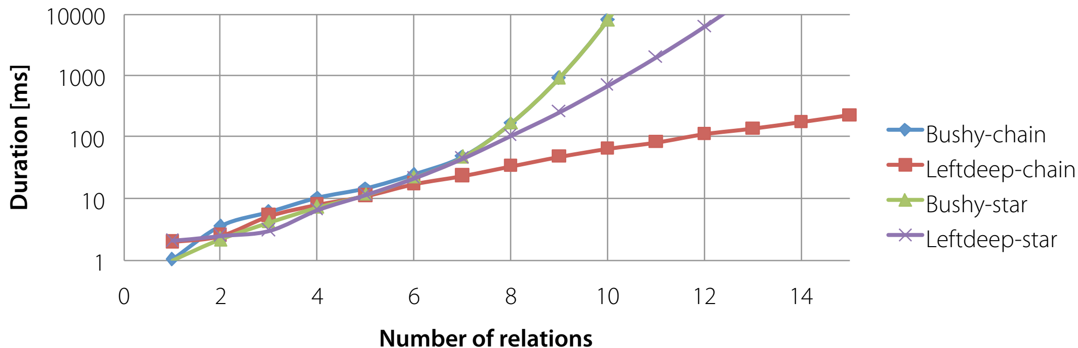


Figure 8.1: Plan generation for chain and star queries, bushy and left-deep, average of 100 runs.

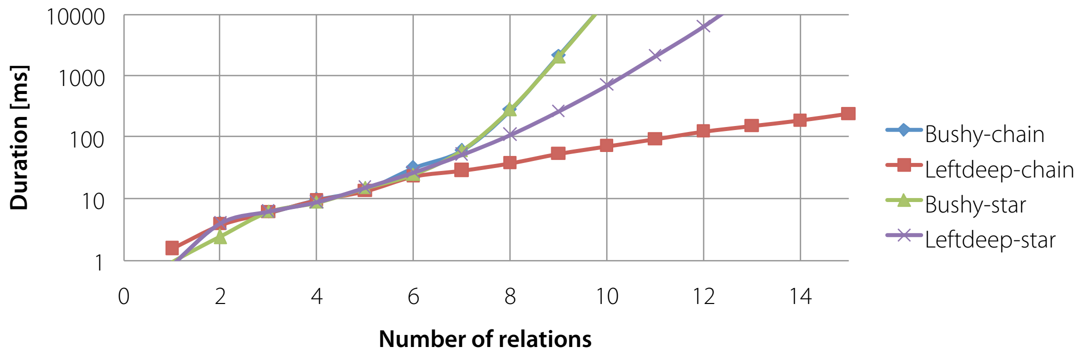


Figure 8.2: Same as Figure 8.1, without reachability caching.

relations are joined in a chain, i.e. 0-1, 1-2, 2-3 and so on. When not considering cross products, the former takes longer to optimize, since the number of valid orderings is greater than for chain joins. The queries optimized only include joins. A real world query is likely to include selections, grouping and orderings, which will increase the size of the search space and thereby the time spent optimizing.

Since these are simple non-comparative timings, we only present the average of 100 runs. We measured standard deviations and found them to be insignificant. The benchmarks were run on an Intel Core Duo T2500 (2×2.00 GHz) with 3 GiB DDR2 memory. The optimizer is single-threaded, so only one core was utilized.

The plots clearly show that bushy plans have higher complexity and take longer to optimize than left-deep plans. Furthermore, it also shows that star queries are more complex than chain queries, as expected. We consider anything much above one second to be too long, which suggests that, at its current state, enumerating bushy plans should be abandoned for >8 relations. Enumerating left-deep plans is feasible up to around 11-12 relations, which also happens to be the default number at which PostgreSQL switches to a genetic algorithm [Pos08a]. This is also the limit mentioned for non-parallel optimizers in [HKL+08], so we are not too far from what is regarded as accepted performance. Queries with > 12 joins are typically OLAP-esque reporting queries where "immediate" response is not expected. MARS' Information Retrieval queries that should respond quickly enough to appear "immediate" to a human user typically have few enough relations to be within the 100 ms range.

Figure 8.2 shows the same timings, but now with reachability caching (as explained in the list in Section 3.6) disabled. We can see that this results in worse run times for bushy enumeration, especially for more than 7 relations. Performance for left-deep plans only marginally decreases (by 50-100ms). This is because left-deep enumeration does not generate as many

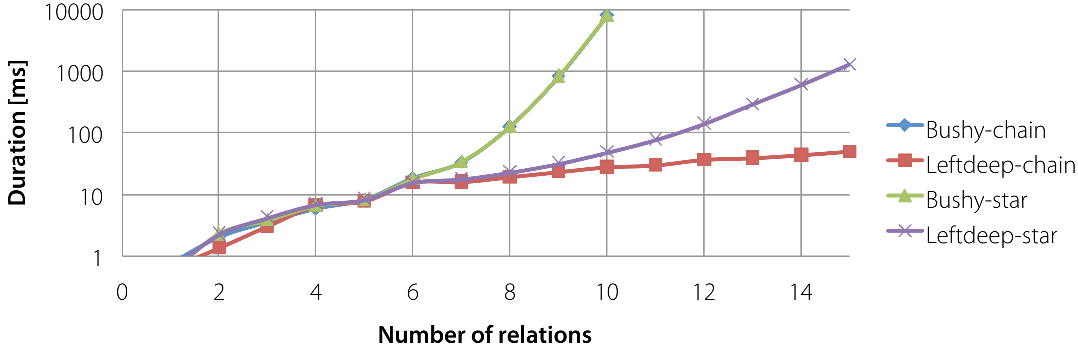


Figure 8.3: Performance of last year's optimizer, without support for sharing and orderings.

unreachable plans as bushy enumeration does, so the check is not invoked that often anyway.

For completeness, we include the plot from last year's report [BN08][1] in Figure 8.3. Then, the optimizer supported neither sharing nor orderings and groupings. Compared to Figure 8.1 we see that adding support for these has not gravely deteriorated the performance. The left-deep *star* joins have been most affected. It is the preparation time of the ordering manager that makes up for most of the increase. This is because the star joins enable a lot more potential orderings than the left-deep do, so the state machines become vastly bigger . The runs doing *bushy* join spend enough time enumerating plans for planning time to dwarf the ordering manager's preparation time, so the increase is not as noticeable there.

Since our design is based on [Neu05], it is interesting to compare our performance to what was reported there. He does not explicitly state it in the context of the graph, but later in his report he mentions experiments run on "a 2.2 GHz Athlon64 system running Windows XP". Neumann has only looked at bushy plans, and his results can be seen in Figure 8.4. He reports around 1 ms run time for 7 relations, increasing exponentially to around 40 ms for 9 relations. His graph shows a straight line on a logarithmic scale, while ours looks exponential even on a logarithmic scale. We see around 40 ms for 7 relations, increasing (more) exponentially to 730 ms for 9 relations. In other words, we see exponential increase in both graphs, but ours come two relations "earlier" and increases faster. Also note that bottom up-plan generation is reported to be faster for a large number relations. This is why we suggest investigating this in Section 9.1.6.

This does not look too good on our part, but we believe there are multiple reasons. First, Neumann's implementation is probably more optimized and refined than ours. He may also do some logical optimizations (as opposed to just code optimizations) we do not. We have not spent much time optimizing our solution, as we wanted to get the design principles right first. For code optimizations, we believe we have potential for improvement in the order component, as it is currently written with functionality and not performance as the primary focus. For logical optimizations, we believe we have potential for improvement in minimizing unreachable plans as explained later in this chapter.

Second, we only do very basic cost-based pruning, and can probably improve our solution here. Neumann describes several techniques in this area (and is likely to have implemented some of them), but we have not had the time for this. This would enable the optimizer to abort searches that is obviously going to return more expensive plans earlier.

Third, we do not know what kind of implementation that was benchmarked in [Neu05], so we do not really know what we are comparing with. It could be anything from a full optimizer

[1]Last year's benchmark was run on the same hardware, but with Vista instead of Windows 7. We do not believe this to have any significant impact.

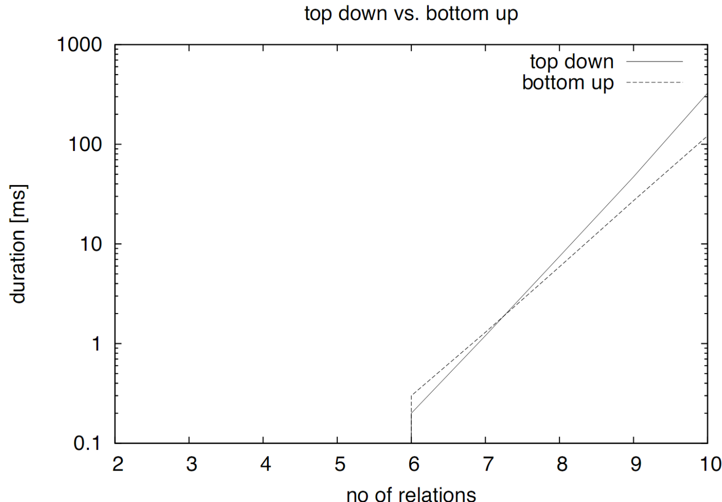


Figure 8.4: Plan generation performance as reported in [Neu05].

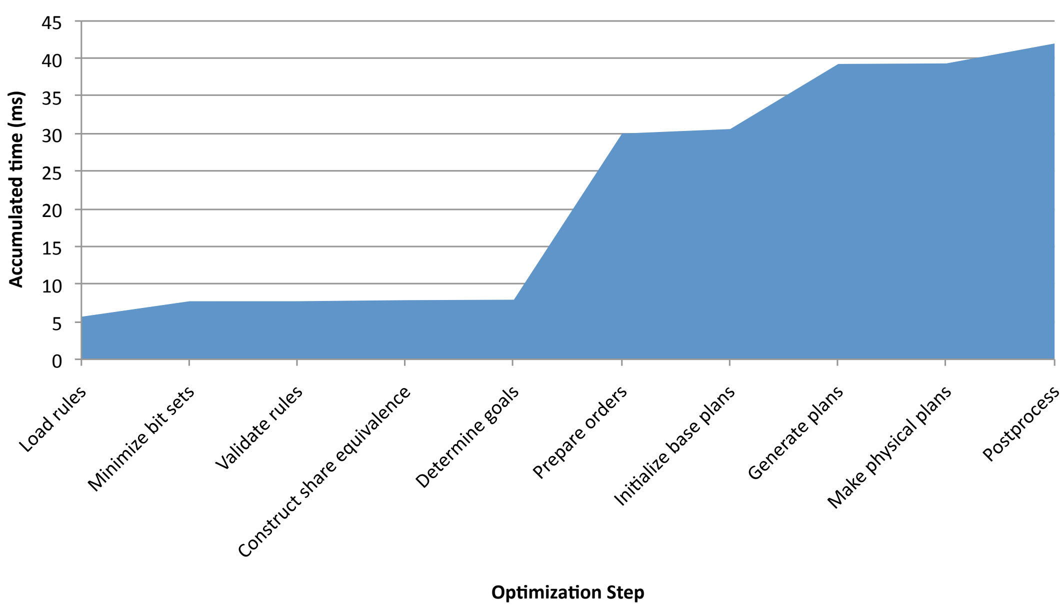


Figure 8.5: Cumulative time consumption, 6 relations

as in our case to only a skeleton.

We have not measured memory usage accurately, but by inspecting the memory use of the optimizer process while running, we see approximately 50 MiB for the largest queries (9-15 relations), less for the smaller.

We also include a visualization of where time is spent during optimization. Figure 8.5 shows cumulative time consumption for the different stages during optimization for 6 relations, bushy chain joins, while Figure 8.6 for 8 relations. As can be seen, the time spent on plan generation increases rapidly for increasing number of relations. This is expected, as this step has exponential time complexity. We also observe that preparing orders is taking a significant amount of time. We probably have a lot of potential for improvement here, as we have spent very little time optimizing this component. Apart from that, the other steps execute fairly quickly.

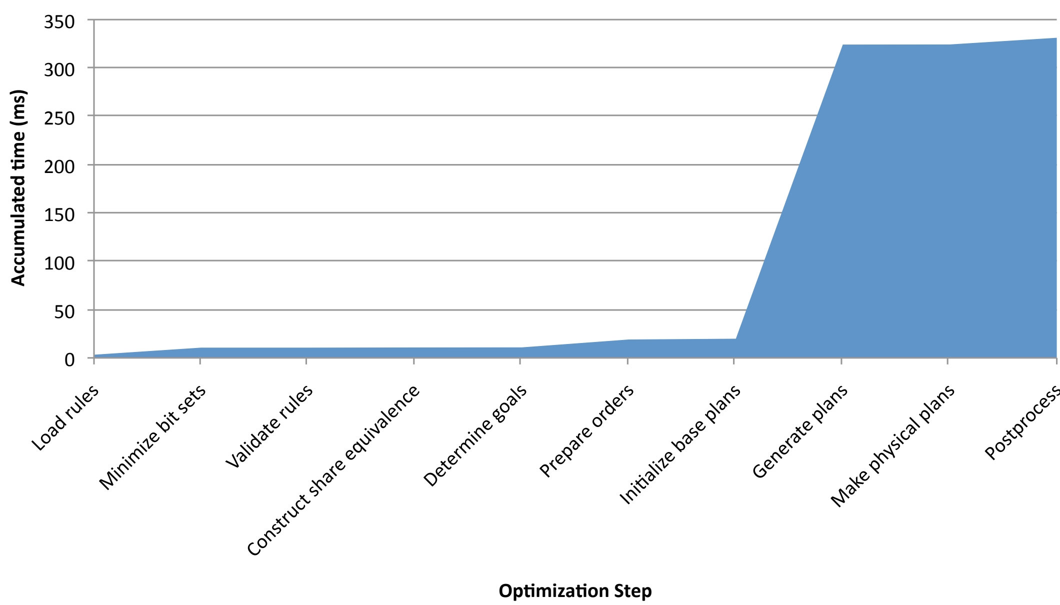


Figure 8.6: Cumulative time consumption, 8 relations

Profiling

To get an overview of where in the code the bulk of the time is spent, as well as to identify any performance bottlenecks, we have done a few profiler runs of the optimizer in action. This is important to be able to optimize existing code, but also helps writing performant code in the future.

Figure 8.7 shows a sample run using the ANTS Profiler [Red08] when performing join ordering of 9 relations. The figure lists the methods where the most time was spent during execution, along with their hit counts. The first three lines are the entry path into the optimizer, and is not that interesting. On lines 4-6 we can see that most of the time was spent in *QueryOptimizer.GeneratePlans* and *JoinRule.(Internal)Search*. This is expected, as it is between these two methods the plan enumeration occurs. We can also identify that *GeneratePlans* and BitSet operations are among the most frequently called methods in the system.

As an example, early profiler runs showed that a significant amount if time was spent in our BitSet implementation. By making BitSet operate on whole ints (32 bits) instead of single bits, we were able to almost double the performance.

8.3 Identified Issues and Suggested Solutions

8.3.1 Exhaustive Enumeration

Currently, our optimizer is close to exhaustive within the limitations set on the search space, and it only employs very limited cost-based pruning. Currently we can control whether bushy or only left-deep plans are considered, but this choice is currently hard-coded in the source code. At its current performance, the optimizer becomes to slow for $\geq$8 relations for bushy plans and $\geq$11 for left-deep plans (where too slow is $\geq$1s).

According to our *fast*-representative, more than eleven relations are not too common in MARS queries. Nevertheless, we want to look into ways of addressing this problem.

Method Name	Time (%)	Time With Children...	Hit Count
Program.Main(string[] args)	0,000	99,937	1
QueryOptimizationSmokeTests.TestBushyChainJoins()	0,000	99,742	3
QueryOptimization.Optimize(OperatorNode query)	0,000	99,725	3
QueryOptimization.GeneratePlans(OperatorNode mu...	0,000	98,854	3
QueryOptimization.GeneratePlans(BitSet goal, ICost ...	5,394	98,847	96 427 263
JoinRule.Search(PlanSet planSet, ICost limit)	0,032	98,840	107 031
JoinRule.InternalSearch(PlanSet planSet, ICost limit,...	6,029	98,836	107 031
QueryOptimization.GeneratePlans(BitSet goal, ICost ...	2,512	96,199	96 411 792
BitSet+BitSetWalker.MoveNext()	35,681	35,681	96 355 440
BitSet.op_BitwiseOr(BitSet a, BitSet b)	12,382	15,114	96 427 377
QueryOptimization.GoalIsUnreachable(BitSet goal)	3,334	10,993	2 366 880
BitSet.Equals(object obj)	8,200	8,200	94 060 279
BitSet.GetHashCode()	4,631	4,631	98 793 511
BitSet.op_LessThanOrEqual(BitSet a, BitSet b)	3,120	3,120	48 119 981
BitSet..ctor(BitSetManager manager, int[] data, int l...	2,742	2,742	96 758 529
UnaryRule.Search(PlanSet plans, ICost limit)	0,003	1,817	15 468
BitSet+BitSetWalker.get_Current()	1,566	1,566	96 248 264
AbstractBaseRule.get_Filter()	1,168	1,330	35 503 803
BitSet.Or(BitSet other)	1,308	1,308	14 634 206

Figure 8.7: Profiler run of a 9 relation bushy star join

Suggested Solution: Heuristics and Cost-based Pruning

One obvious idea is to make the choice between bushy and left-deep plans at runtime based on the query complexity, but this can lead to suboptimal plans. It is probably a better idea to try and increase the performance of bushy enumeration.

We see cost-based pruning as a good way to do this. The idea is to do incremental costing during plan generation, all the way maintaining a upper cost bound equal to the best plan found so far for a given sub-problem. Whenever a new exploration task exceeds this bound, it can not possible yield a better plan and is aborted. Currently, cost-based pruning only happens in `PlanSet` — at which point the plans are already constructed. The idea is to continuously decrease the limit to prevent exploring more expensive plans. This is non-trivial when also considering sharing opportunities, which is why we have not implemented it.

For basic cost-based pruning, *infinity* is used as the initial bound. A better strategy is to try and achieve a tighter cost bound to start with. The simplest way is to use the cost of the canonical plan (the query directly translated into a operator graph). A better way is to employ various heuristics to quickly construct a much better plan than the canonical plan and use the cost of this plan as the initial cost bound.

8.3.2 Unreachable Plans

Consider the query in Figure 7.11, which joins four relations. Following the process described in 3.6.4, the global goal of this query will be determined to be `Goal` $= \{\text{Id0}, \text{Id1}, \text{Id2}, \text{Id3}, \bowtie_{0,1}, \bowtie_{1,2}, \bowtie_{2,3}\}$. We have ignored the NameX properties. The topmost join has the predicate `[Id2] = [Id3]`, and therefore `Produced` $= \{\bowtie_{2,3}\}$, `RequiredLeft` $= \{\text{Id2}\}$, `RequiredRight` $= \{\text{Id3}\}$.

Following the algorithm for the join rule, described in 5.3.9, the rule will try to satisfy `Goal` by exhaustively splitting `wantedProperties` $=$ `Goal` $-$ (`Produced` $\cup$ `RequiredLeft` $\cup$ `RequiredRight`) $= \{\text{Id0}, \text{Id1}, \bowtie_{0,1}, \bowtie_{1,2}\}$ between the left and right input. It will therefore, for instance, at

one moment try to get $\{\text{Id2}, \bowtie_{0,1}, \bowtie_{1,2}\}$ as its left input and {Id0,Id1,Id3} as its right. Obviously, this is not possible since $\bowtie_{0,1}$ and $\bowtie_{1,2}$ requires Id0 and Id1 and the plan generator will immediately return with no plans. Still, it takes a significant amount of time to try it, especially when the number of operators and properties become large.

Suggested Solution

To prevent this from happening, we could pre-compute transitive closures on the required properties of all rules instantiated during the preparation phase. This could be implemented as an array with length equal to the number of bit properties, where element i contains the minimum set of bit properties required to produce property i.

For instance, for the example above, the array entry for $\bowtie_{0,1}$ would be {Id0,Id1}. This array would be used by the different rules to ensure that they never ask for something that is guaranteed to be impossible, thereby saving time. For instance, the join rule would consult this array before iterating over all the different ways of splitting the `wantedProps` bitset.

8.4 Extensibility

As an example of how extensible our optimizer design is, we include an amusing example from when we first managed to integrate our optimizer with MARS and optimize a query on the fly. MARS does not provide any interface to see the operator graph of the query that is run, so we only had our own debug output indicating that it was the optimized query that was being run. Furthermore, the data set was too small to notice any difference in execution time at the time, so we really did not know if MARS actually executed the optimized query and not the input query. We came up with the idea of having the optimizer deliberately leave traces within the optimized query. We quickly implemented a post-processing rule that inserts a map operator in the query, adding an extra column to the output, named "OptimizerWasHere". The output from a sample query is:

```
| DocumentId | DocumentName                      | OptimizerWasHere |
-------------------------------------------------------------------
| 0          | /data/docfeeder/newdocs/0020_c | Oh yes, I was!   |
|            | 2976_drugsafety_wp.pdf.xml     |                  |
```

To illustrate how easy it was to implement this post-processing rule, we have included the *full* source code in Listing 8.1. No configuration changes or changes to the optimizer were necessary — it automatically picks up new rules.

8.5 Important Missing Features and Implementation Ideas

Even though there are a lot of things we have not had time to study in depth and/or implement, some unimplemented features are so important we describe the feasibility of implementing them — as well as briefly sketching how we envision them implemented.

8.5.1 Compound Rules

The benefits and caveats with compound rules are described in Section 3.9. Most of the infrastructure for implementing this is already in place. In particular, rule binders are invoked via graph patterns, and not via a one-to-one mapping of operator types.

```
[Postprocessor]
public class OptimizerWasHere : AbstractTransformationRule {
   public override AbstractNodeMatcher Pattern {
      get { return Match.NodeOfType("OutputOperator").GroupAs("output"); }
   }

   public override bool Fire(GraphSearcher graphSearcher) {
      OperatorNode outputOperator = (OperatorNode)graphSearcher.Groups["output"].Single();

      //We change outputOperator.Children with InsertParent, so copy the list
      foreach (OperatorNode query in new List<Node>(outputOperator.Children)){
         OperatorNode mapNode = new OperatorNode(typeof(MapOperator));
         mapNode["RetainInput"] = true;

         mapNode["ParameterMap"] = new Dictionary<IdentifierProperty, ExpressionProperty>()
            { { new IdentifierProperty("OptimizerWasHere"), new ExpressionProperty("\"Oh, yes, I was!\"") } };
         query.InsertParent(mapNode);
      }
      return true;
   }
}
```

Listing 8.1: Example Transformation Rule

As opposed to the rules we have developed so far, which are inherently oblivious of the semantics of *other* rules, compound rule binders have to know about the semantics of all the rules they encompass. To keep a sensible separation of concern, this suggests that they should be completely separate rules with their own rule binders — i.e. the join rule should *not* know about all the possible ways to compound. Instead, there should be something like a SelectCompoundRuleBinder, which knows when it makes sense to lock a Select in place — be it over a Lookup or a Join.

Compound rules can possibly come in many forms. In Section 3.9 we had a simple example of a Select over a Lookup. That rule is a leaf node, and it has only one output. However, compound rules might just as well be internal nodes with multiple inputs with differing requirements, such as when the bottom node of the compounded sub-graph is a join.

Compound rules might also contain other compound rules, which in turn might contain others. This is not as far-fetched as it may first sound. For example, a compound can be made with a Map over the compound of a Select over a Lookup. Since pattern matching is done breadth first, this should not be too hard to implement. However, it requires some changes to how rule binders are invoked. When the pattern of a compound rule binder matches a sub-graph, further matching on that sub-graph should be directed by the rule binder.

We list a few examples of compound rules that are desired. Note, however, that in addition to the patterns declared by the rule binders, some logic must also be applied to find out whether the rule should be applied at all. Furthermore, it might be beneficial to have some auxiliary preprocessors that properly reorder some operators, to keep patterns and rule binders simple — such as moving a Selection-operator below a Map

- A Selection over a scan such as a Lookup, if its placement there can be ascertained even before costing.
- (O)Near over Lookups.
- ScoreOccurrences over one or more (O)Near- and Lookup-operators.

- A Selection or a Map over a Join, when the predicate is expensive. See next section

As explained in Section 8.5.4, however, resorting to compound rules is not necessary to simply restrict movement of a single operator.

8.5.2 Projections

Projecting away attributes is handled by MARS' Map-operator, as is the evaluation of new and aliasing of existing attributes.

With the exception of map operators with expensive expressions, e.g. expensive user-defined functions, a greedy strategy will yield a good result for map operator placement in most cases. Map is therefore a good candidate to be taken out of the plan generation step and into post-processing to reduce search space size. We therefore completely ignore maps during the search phase, and apply removal and/or renaming of the attributes as a post-processing step as the topmost operator in the query. Only Map-operators that evaluate new attributes (e.g. Score $=$ 70 Score $+$ 30 ProxScore) are kept, but locked as much in place as possible.

Unfortunately, we have not had the time to implement the greedy strategy and the optimizer therefore only inserts projecting maps as the topmost operator. It still handles all queries correctly, but it can lead to sub-optimal plans when the attributes that can be projected away earlier make up a substantial amount of memory — especially if temporary relations must be flushed to disk during sorts or hash-joins.

Anyway, we outline the greedy strategy here:

1. (already implemented) All map operators that do not produce attributes are removed during pre-processing.
2. (already implemented) Map operators that do produce attributes are locked in place just before the first operator that needs them in the search phase. Compound rules could also be used for this purpose.
3. (not implemented) During post-processing, all pipeline-breaking operators (operators that break the data pipeline and materialize data, e.g. hash join and sort) and their inputs are examined. If there are attributes in the tuple stream that are not needed past this point, the size of the data that can be projected away is estimated. If this amount is over a certain threshold (the map is not free, either), a map operator is inserted to project away the attributes before the pipeline breaker.
4. (not implemented) During post-processing, successive map operators with none or only non-pipeline-breaking operators between them are merged. For example, if a rename operator can be merged with a projecting or attribute producing Map, this is preferable — the fewer operators, the better.
5. (already implemented) During post-processing, if needed, a final map operator is inserted as the topmost operator of the query to reorder, remove or rename attributes to make the query output schema correct.

8.5.3 Expensive Predicates and User Defined Functions

As briefly described in Section 1.6.7, user defined functions used either by Map-operators to evaluate new attributes or in predicates in Select-operators might be expensive. In the case of the Select, where the predicate is expensive, we will want to evaluate the expensive predicate

after other operators or predicates have reduced the result set as much as possible. An example was given in Section 1.6.7.

The same is true for Map-operators with expensive evaluations. Additionally, if no operators reduce the result set, we want to evaluate the expensive expression *before* any result set increase — to avoid computing the same expression multiple times. This problem can be mitigated with memoization in the operator, though.

However, to know how a join affects the record set, we need the functionality described in Section 8.5.6.

To implement this, more work on the dependency resolving must be done. Specifically, we want to add a dependency from the expensive selection to other set-reducing operators — and from the set-increasing operators to the expensive map operators. Exactly how these dependencies are devised have not been studied, however.

8.5.4 Dependency Mapping of Outer- , Anti- and Semi-Joins

Outer-, anti- and semi-joins require special attention because they are not freely reorderable — i.e. they are not always associative with each other. Consider for example[2] $R ⟕ (S \bowtie T)$ vs. $(R ⟕ S) \bowtie T$, where $⟕$ is a left outer join. With the input relations shown in Table 8.1a–8.1c we get the outputs shown in Table 8.1d and 8.1e, illustrating the semantic difference. The corresponding join graphs are shown in Figure 8.8. Similar examples can easily be devised for anti- and semi-joins, but we omit them for brevity.

To constrain the movement of the joins correctly, we need to add enough dependencies to the operators. For example, if $R ⟕ (S \bowtie T)$ is the input query, we would add a dependency from $⟕_R$ to $\bowtie_{S,T}$ to "lock" the ordering. Of course, in this simple example query, this leaves no other possibilities.

Exactly how to devise the correct dependencies have not been studied in depth. However, we did find an algorithm described in "Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering", written by J. Bruce et. al [BGPS01]. We believe the algorithm can be implemented as a pre-processor without too much hassle. The gist of the algorithm is to iterate through the operator graph in topological ordering, to build "Extended Eligibility Lists" or "EELs". EELs contain a list of tables referenced by the join predicate — from which dependencies can be derived.

The algorithm is clever enough to also consider predicates' "NULL-tolerance" when building the lists. A NULL-intolerant predicate is one which never evaluates to true when a referenced attribute is NULL. With this, we can possibly rewrite outer joins to inner joins, and consequently relax the dependencies.

To summarize, dependencies will be used to prevent erroneous movement of these kinds of joins, but we have not had time to find out exactly how the dependencies are devised.

8.5.5 Equivalence Class Joins

Figure 8.9 shows two possible join-trees for the star-join query $A \bowtie B \bowtie C \bowtie D$. We assume all relations are joined on the same key id. When the rule binder for joins instantiates rules it encounters a problem with regard to what rules should be instantiated, and what their dependencies should be. For example, if we only instantiate one rule for the four-way join, we will be unable to move any of the joins around. What should the pairs be — i.e. which two relations should be the input to the "bottom rule"? In Figure 8.9a, the limitations are clear if we say that

[2] example adapted from [BGPS01]

tid	a
r1	1
r2	3
r3	5

(a) R

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

(b) S

tid	b
t1	1

(c) T

r1	s1	t1
r2	NULL	NULL
r3	NULL	NULL

(d) R ⟕ (S ⋈ T)

r1	s1	t1

(e) (R ⟕ S) ⋈ T

Table 8.1: R, S and T — and two different outer join orderings

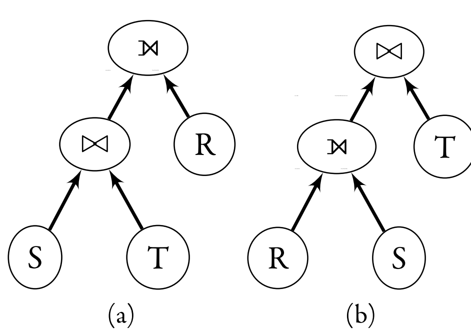


Figure 8.8: Join graphs corresponding with Table 8.1

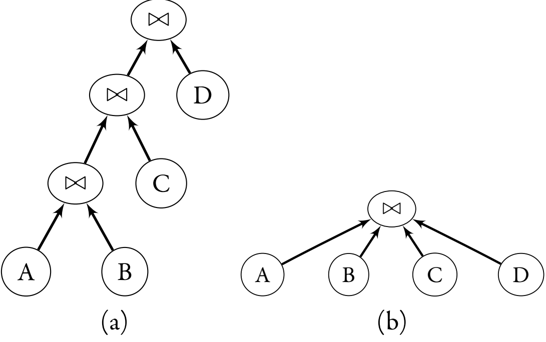


Figure 8.9: Two equivalent four-way joins on the same key

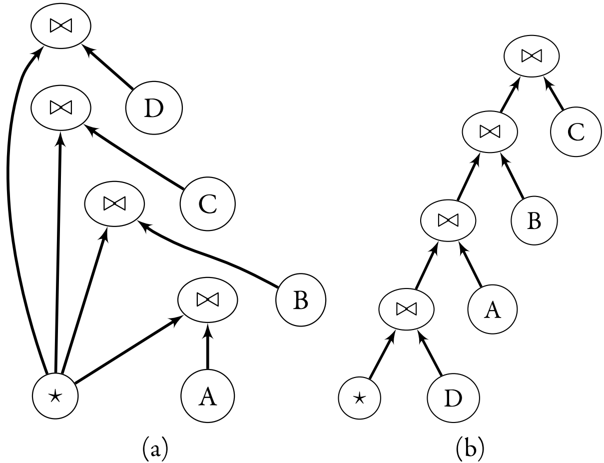


Figure 8.10: Rule instances

the $\bowtie_D$-rule has to depend on its left input — we would then be unable to reorder it into e.g. $(D \bowtie A) \bowtie B \bowtie C$.

This limitation is present in the current join rule binder, and this section describes how we imagine a proper implementation. It is not a problem specific to our design, but something all implementations of join orderings have to deal with. Without loss of generality, we limit the discussion to left-deep joins.

In Section 5.2.1 we described how equivalent attributes are mapped. These maps are available by the time of rule instantiation. Thus, the join rule binder can identify that the join keys of A…D are all equivalent — i.e. that it does not really matter which relation any of them are directly joined with, as all are valid.

To remedy this, we imagine that instead of having $n - 1$ rules where the "bottom rule" is instantiated with two relations as required inputs, we have n join-rules where the required left input for all of them is the equivalence class representative $\star$ — as shown in Figure 8.10a. We also need to add a base-relation for the representative. Figure 8.10b shows an example of how the rules can be combined to represent the join-ordering $((\star \bowtie D) \bowtie A) \bowtie B \bowtie C$. The join rule then treats $(\star \bowtie D)$ as a special case both when costing and when reconstructing the query graph — it is equal to D, so the resulting graph has three join operators.

Since we have not had time to implement this, we have not yet decided whether there should be a MultiJoinSplitter-pre-processor that splits multi-joins into multiple operators — as described in Section 5.2.1 — or if it should be the task of the join rule binder to instantiate multiple rules for multi-joins.

8.5.6 Reasoning on Relation Semantics

Key properties and -relations in a database are used to enforce constraints. However, they can also be used to reason whether joining a relation on a specific key will increase, decrease or preserve the number of records. For example, if we have foreign key from person.city_id to city.city_id we can be certain that joining in city will preserve the number of records. However, if city is the relation we join person to, we cannot know whether the record set is preserved, increased or decreased — as there could possibly (albeit unlikely) be exactly one person per city, (more likely) multiple people per city, or none.

With outer joins, however, we can be sure that the number of results either remains the same or increases — the latter can possibly be reasoned as more likely with 1-to-many or many-to-many relationships.

Reasoning that a join reduces the amount of records, however, is not possible to deduce from foreign key relationships alone. It is possible that augmenting the catalogs with additional information about relationships, as suggested by the paper briefly covered in Section 1.6.10, is a good approach, but we have not studied this further.

However, as inferring that the record set will remain the same is possible and opens a wealth of new optimization opportunities, it is desirable to implement it. A motivating example is given in Section 3.4. With foreign key relationships available in the system catalog, the join rule binder only needs to have its *SetBehaviour*-method check that the join key is a foreign key pointing *to* the relation being joined.

9 Further Work and Conclusion

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

—Brian Kernighan

9.1 Further Work

In the last section of the previous chapter, we wrapped up with ideas of features we have a reasonable idea of how to implement.

However, there are many areas we have not investigated at all, with ripe opportunities for future improvements.

In this section, we summarize further work for this thesis. Some of it, especially the first four sections, is work that needs to be done before the query optimizer is production-ready. The last five sections include possibilities it would be interesting to pursue, but is not critical.

9.1.1 Tighter Integration with MARS

Although we have successfully integrated our optimizer with MARS, there are still several tasks in this area that need to be addressed. We list some of them here.

Logical operators. Currently, MARS only has physical operators (i.e. HybridHashJoin and MergeJoin, not Join). This does not allow for a truly declarative query language. Currently, we transform HybridHashJoin and MergeJoin to a logical join internally in the optimizer, but ideally MARS would support a Join-operator.

System catalog. A system catalog is needed to look up information on relations and indexes and create, store and look up statistics. Currently, MARS has very limited support for this, so work needs to be done in this area. Some improvements are suggested in Section 4.3.2.

Plan caching in a search engine is arguably different to that of an RDBMS. First, the contents of the indexes change very rarely, comparatively. Thus, if the exact same query is performed in locality of time, layers above the evaluation engine will likely cache the entire *result* of the query — which is not prohibitively expensive when only caching e.g. the 100 first results. Secondly, parametrized plans are less likely to be reasonably close to the optimum. The selectivities of the terms in a full-text query can vary widely — as opposed to OLTP-esque lookups on primary keys.

One avenue of pursuit is perhaps to research whether the memoization table can be reused for subsequent or parallel optimizations. For example, if a sub-plan is memoized with parameters that have selectivities within certain limits of the current query, then that sub-plan could be considered reused — saving planning time. However, not only would this require drastic changes to the optimizer, but without knowledge of realistic workloads, it is a solution looking for a problem.

Another approach is to have a data structure that given a parametrized plan returns a list of plans for various input selectivities. Then, the existing plans can be examined to find out how close the assumed selectivities are to the input parameters. If they are within a certain range, the plan can be reused.

9.1.2 Operators

Support for more operators needs to be implemented to enable the optimizer to handle more advanced queries. This includes operators such as And, Or, and Trim. We also need to implement predicate splitting as described in 3.9 to enable more advanced selection optimizations.

Ultimately, a clear API and comprehensive documentation should enable developers of future operators to implement the necessary rules and their binders without detailed knowledge of the optimizer's inner workings.

9.1.3 Support the Exchange Operator

The exchange operator is important with regard to parallelizing execution over multiple nodes. Since we were told by our *fast*-representative to not focus on it, and we have been unable to execute and test such queries anyway, we have not studied its implications in depth. However, we have had it in mind — such as when we mention communication costs in the chapter describing the cost model.

9.1.4 Better Cost Model

The currently implemented cost model is relatively generic and is not accurately adapted to MARS. Nor does it model memory usage, network I/O cost, parallelization opportunities and so on. It works as it is, but to achieve good results, quite a bit of time must be invested in accurately cost modeling MARS' operators.

This also includes the use of histograms to determine the selectivities and expected tuple counts for queries and index lookups, as well as using system catalogs do determine tuple sizes.

9.1.5 Optimizing the Optimizer

Except in the innermost loops of the search phase, we have not focused deeply on optimized code and data structures. As shown in Section 7.3, the optimizer is fast enough to be usable, but there is still room for lots of improvements.

9.1.6 Investigate Bottom-up

As explained in Section 2.5.3, we have settled for a top-down approach. [Neu05] says that bottom-up may be beneficial for larger queries, see figure 8.4. It might be wise to spend some time investigating the possible performance gains a bottom-up approach would give, as converting the rules to bottom-up is not too hard. A sketch for the selection rule bottom-up is given below.

```
1 public override void SearchBottomUp(PlanSet plans, BitSet current) {
2    foreach (Plan inputPlan in plans) {
3       Plan selectionPlan = new Plan(inputPlan) { Rule = this };
4       plansCache[current | Produced].AddPlan(selectionPlan);
5    }
6 }
```

Listing 9.1: Imaginary *SelectionRule.SearchBottomUp()*, very simplified

9.1.7 Opportunities for Planner Parallelism

Currently, we cannot do much over 11–12 relations within a feasible amount of time on one thread without limiting the search space. This can lead to suboptimal plans.

The current trend in processor development, is to add more cores, and to use more processors. The raw clock speed no longer increases substantially. Because of this, it can be smart to look at parallelism. The most prominent problems with parallelization are data dependencies and distributing work items between threads. [HKL+08] has achieved close to linear speedup for dynamic programming of join ordering and has managed to increase the number of relations for bushy up to approximately 16 within the second.

Our algorithm is top-down, so the method used in [HKL+08] is not directly applicable, but we still have some ideas on how to do it. Instead of waiting for the plan generator to produce each plan, sub-plan requests can be queued to a thread pool and the requester notified when it is ready. Pre-/post processing steps are harder to parallelize because of the linear nature, but they are not the greatest contributions to optimization time anyway. Also, it is not too complex to convert our optimizer to a bottom up one, which should make it easier to apply the method used in the above paper.

9.1.8 Parallel Execution and Costing

It is not only the optimization of query plans that can be parallelized — the execution of them can as well. For example, sorting can be parallelized, or different parts of the query graph can be run on different threads. Parallelism does not only constrain itself to a single processing node. It is quite common to employ parallelized execution between nodes in a cluster for search engines. MARS also supports this.

To create parallel plans, the optimizer must both know which operations can be parallelized, and the cost model must honor such plans. However, the cost model may change if the system is under heavy load, and there is no gain in parallelizing the query, since all nodes are swamped anyway.

9.1.9 Compensating Operators

By utilizing compensating operators when sub-problems cannot be completely shared, but is close enough so that large parts can be reused, DAGs can be created when share equivalence is a too strict requirement. We briefly mentioned this in Section 3.7 and gave an example.

Such a feature needs a reasoning framework that extends and encompasses the current share equivalence model to keep track of what compensations are possible at what cost. However, we have not had the time to study this in depth and leave this as an open possibility.

9.2 Evaluation and Conclusion

In the introduction, we stated multiple goals. The first one was to *extend the optimizer with new rules to have it support a wider set of MARS' algebra — at least lookup, joins, sorting and grouping.* We have spent most of the spring on implementation, and the optimizer now supports lookup, the score occurrences operator, selection, projection (map), merge and hash joins, sort operators, and streaming- and hash group. This is enough to express the most common queries, and the optimizer can now optimize such queries.

The next goal was to *implement native support for DAG-structured query plans and multi-queries*. We did not have time for this in the specialization project, but since the design was prepared for it, we were able to implement this without too much difficulty. The optimizer is able to analyze the input query for sharing possibilities, and create a query DAG where the cost model deems it beneficial. It can do this regardless of the form of the input query — DAG or non-DAG. We have also seen that DAG-structured query evaluation plans can reduce the evaluation costs significantly, especially for the multi-queries common to MARS.

We also aimed to *implement support for tracking available orderings and groupings in a query plan*. This was needed to reason about merge joins and streaming groups, and we considered this to be a requirement for having a usable optimizer. We solved this by implementing an orderings- and groupings component that handles this logic. This component is actually capable of more fine-grained reasoning (about groupings) than MARS can utilize at the moment, so we exceeded our goal here. It is also very efficient during plan generation, but the initialization procedure leaves a lot to be desired efficiency-wise.

Finally, we wanted to *integrate the optimizer with MARS by injecting it into MARS' query pipeline, enabling end-to-end execution of queries with the optimizer.* This was not a priority due to the current development status of MARS, but we were able to achieve this goal as well. Although we received help from *fast*, some of the work we had to do here resembled detective work at a high level, since we did not have access to MARS' source code. The result is that we are able to intercept and optimize queries between the parsing and evaluation stages. Since MARS does not have the required system catalogs in place, such catalogs were stubbed.

We can therefore conclude the master's thesis with a system where we can run queries end-to-end with optimization enabled — which was the ultimate goal of the specialization project and master's thesis.

We now summarize why we believe our work is of great use to *fast* and MARS.

MARS has no optimizer. Currently, MARS does not employ an optimizer. Queries must be optimized by hand, directly using physical operators. By implementing an optimizer, we can do this automatically and hopefully better. Furthermore, the users can focus on writing readable and maintainable queries, instead of optimized ones.

Declarative queries. An optimizer enables the user to write queries that are truly declarative in any language, for example SparQL, not only MQL (MARS' physical query language).

Integration with MARS. We have been able to integrate the optimizer with MARS, proving that it plays well with the implementation of MARS and its algebra. Both the optimizer and MARS are implemented in C# on .NET.

DAG support. MARS supports DAGs, and our optimizer will construct DAG plans where beneficial. Since this is a defining feature not commonly found elsewhere, it should make this work even more interesting. We also support optimizing multi-queries, which can drastically reduce the evaluation costs.

Extensibility. *fast* wanted an optimizer that is extensible, to support future operators. Our optimizer is extensible in terms of both operator rules and the cost model.

Separate component. The optimizer is not tied closely to MARS and can without too much work be adopted to another system *fast* may develop in the future.

Summarized, we would say we have reached the goals stated and are satisfied with the result. Not only do we have a running optimizer integrated into MARS, but we have also documented the important parts of its inner workings. It is still far from production-ready, but should provide a very good foundation for *fast*'s further work with an optimizer for MARS. Since we have worked with this optimizer for almost a year now, we have some very good ideas about what should be the next steps and outlined them in the previous sections. The optimizer is, of course, far from perfect.

During the development of our optimizer we have several times thought "Why did it do *that*, it cannot possibly be right!?" — especially for complex queries with complex functional dependencies. Sometimes the reason was bugs in the optimizer, but we have more than once concluded with "Aha, *that* is why!" when discovering an optimization we missed but the optimizer found. We believe that is a good thing.

9.3 Contributions

We hope we have been able to sufficiently and humbly communicate that we have based our work on that of Dr. T. Neumann and G. Moerkotte, as pointed out in Section 2.6. Without their work, we doubt we would have got as far as we have.

However, as pointed out in Sections 3.6.7 and 6.3.5, we have made several improvements here and there, and throughout the thesis we put the architecture in many contexts not described in their works — such as optimizing multi-queries.

At any rate, we believe the most significant contribution from our master's thesis is proving that the architecture not only works well — it can also be *implemented* well enough to be of considerable interest for *fast*'s future query optimization needs in MARS, and possibly other projects.

A Runtime Output

> *"No matter how slick the demo is in rehearsal, when you do it in front of a live audience, the probability of a flawless presentation is inversely proportional to the number of people watching, raised to the power of the amount of money involved."*
>
> —Mark Gibbs

To give the reader an introduction to the look and feel of MARS and our optimizer, we have included an example of what is looks like when executing a query against MARS with our optimizer enabled. The only interface against MARS that is available to us is a console interface, so in the following example, we show the interaction with MARS through the console. We have edited to output somewhat to make it fit inside the report.

We start by booting the MARS node:

```
Microsoft Windows [Version 6.1.7100]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Mars.Net>NodeRunner.exe --shell
DefaultNode>
```

Next, we enter the *QueryFeeder* component to execute queries.

```
DefaultNode> i QueryFeeder1
*** Welcome to MARS Query Feeder ***

Query>
```

MARS is now ready for our query. We input a simple query searching for "fredrikstad", grouping the hits by document type and listing the hits by name.

```
Query>Output(;\
     >   Group(sorted=False,groupFields=["Input1.DocumentType"],\
     >          aggs="Count"=Count(args=Null,filter=Null);\
     >       MergeJoin(joinPrefix=1;\
     >           Lookup(word="fredrikstad",index="Occurrence1"),\
     >           Lookup(word="DocumentType",index="Occurrence2")\
     >       )\
     >   ),\
     >   MergeJoin(joinPrefix=1;\
     >       MergeJoin(joinPrefix=1;\
```

```
>           Lookup(word="fredrikstad",index="Occurrence1"),\
>           Lookup(word="DocumentType",index="Occurrence2")\
>        ),\
>        Lookup(word="DocumentName", index="Occurrence2")\
>    )\
> )
```

This yields the following output. First, some timing statistics from the optimizer is printed, followed by the query results. As can be seen, the "OptimizerWasHere" rule mentioned in Section 8.4 is enabled. Note that the timings below are wall-clock timings and include just-in-time compilation overhead and the like.

```
Optimizing...
Time spent preprocessing: 00:00:00.0009095
5 base rules and 4 search rules loaded.
Time spent loading rules: 00:00:00.0027660
Time spent minimizing bit sets: 00:00:00.0015886
Time spent validating rules: 00:00:00.0000077
Time spent constructing share equivalence classes: 00:00:00.0003762
Time spent determining goals: 00:00:00.0000144
Time spent preparing orders: 00:00:00.0246010
Time spent initializing base plans: 00:00:00.0024380
Time spent generating plans: 00:00:00.0044331
Time spent making physical plans: 00:00:00.0001144
Time spent postprocessing: 00:00:00.0025216
Total time spent optimizing: 00:00:00.0543951
Total memoization entries: 18. Actual plan sets: 10, actual plans 10.
Plans generated: 7, Unreachable plans: 6, plans returned from
    memoization table: 12, rule appliances: 7
...

Record set name: Primary

--------------------------------------------------
| Input1.DocumentType | Count | OptimizerWasHere |
--------------------------------------------------
| va                  | 2     | Oh yes, I was!   |
--------------------------------------------------
| na                  | 2     | Oh yes, I was!   |
--------------------------------------------------
| ea                  | 2     | Oh yes, I was!   |
--------------------------------------------------
| ua                  | 4     | Oh yes, I was!   |
--------------------------------------------------
```

```
Record set name: Output0

--------------------------------------------------------------------------------
| DocumentId | Input0 | Input0.Input0.Scope        | Input0.I | Input1.Docu | Opti |
|            | .Input |                            | nput1.Do | mentName    | mize |
|            | 0.Posi |                            | cumentTy |             | rWas |
|            | tion   |                            | pe       |             | Here |
--------------------------------------------------------------------------------
| 229770     | 52     | /documents[1]/document[3 | ea       | /data/docfe | Oh y |
|            |        | ]/body[4]/paragraph[7]/s |          | eder/newdoc | es,  |
|            |        | entence[1]/location[1]/b |          | s/doc234901 | I wa |
|            |        | ase[1]:14 Attribute      |          | .xml        | s!   |
--------------------------------------------------------------------------------
| 229770     | 53     | /documents[1]/document[3 | ea       | /data/docfe | Oh y |
|            |        | ]/body[4]/paragraph[7]/s |          | eder/newdoc | es,  |
|            |        | entence[1]/location[1]:1 |          | s/doc234901 | I wa |
|            |        | 5 Text                   |          | .xml        | s!   |
--------------------------------------------------------------------------------
| 239320     | 1099   | /documents[1]/document[3 | ua       | /data/docfe | Oh y |
|            |        | ]/body[4]/paragraph[133] |          | eder/newdoc | es,  |
|            |        | /sentence[1]/location[3] |          | s/doc305151 | I wa |
|            |        | /base[1]:14 Attribute    |          | .xml        | s!   |
--------------------------------------------------------------------------------
...
```

B Code Samples

> *"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."*
>
> —Bill Gates

We do not include the full optimizer code base in the report, as it counts approximately 15,000 lines of code. We refer to the accompanying digital appendix for the full code base. See Appendix C for a description of the contents.

The rest of this appendix includes selected code samples that were too long to include in the main part of the report.

B.1 dot Language Sample

Listing B.1 shows the GraphViz [AT 08] dot language used to visualize the operator graph in Figure B.1. This example includes some TEX-code which is then processed by *dot2tex*. However, most graphs in this thesis are TEX-less and generated with GraphViz. Its declarative language makes it easy to auto-generate graphs. These have been of immense value when analyzing various graph structures throughout our work. We have had to post-process most graphs to make them fit on an A4-page, though.

B.2 Rule Binder Initialization

As explained in 5.1, the optimizer does not know the rules at compile time, meaning that new rules can be added without changing the optimizer core at all. To be able to do this, is uses reflection, which is a feature in .NET for reasoning about program metadata.

To be taken into consideration for optimizing, all the rules have to do is to have a rule binder that declares the `[RuleBinder]` attribute and implements the IRuleBinder interface. Then they have to be placed in an assembly (DLL, .NET equivalent of Java JARs) that is visible to the optimizer.

At optimizer startup, *InitRuleBinders* is called, and all classes in all known assemblies are enumerated. If the class declares the `[RuleBinder]` attribute, it will be saved in a list for later use. Reflection used like this is somewhat expensive, but since this only happens once at system startup, it is not a problem.

Then, during the preparation phase of each query, all the previously found rule binders are instantiated and used for instantiating rules. Note that the optimizer never cares about where or how the rule was implemented.

```
digraph G {
   rankdir=BT;
   edge [style="line width=1pt"];
   ranksep=0.2;
   ordering=in;
   J1 [texlbl="$\join$"];
   J2 [texlbl="$\join$"];
   J3 [texlbl="$\join$"];

   B -> J1;
   A -> J1;

   C -> J2;

   J2 -> J3;
   D -> J3;

   J1 -> J2;
}
```

Listing B.1: dot file used to create Figure B.1

Figure B.1: Example Dot Figure

```
/// <summary>Reflect, find and instantiate rule binders.</summary>
private static void InitRuleBinders() {
   RuleBinders = new List<IRuleBinder>();
   // Add all rulebinders marked with [RuleBinder].
   foreach (Type type in ReflectionPluginHelper.GetTypesWithAttribute(typeof (
        RuleBinderAttribute)))
      AddRuleBinder(type);
}

/// <summary>Instantiate and add the rule binder with the given type.</summary>
/// <param name="type">Type of the rule binder to add</param>
private static void AddRuleBinder(Type type) {
   RuleBinders.Add((IRuleBinder) Activator.CreateInstance(type));
}

/// <summary>Gets all types in the accessible assemblies tagged with the specified attribute.</
     summary>
/// <param name="attributeType">Type of the attribute to look for.</param>
/// <returns>A enumerable of the found types.</returns>
public static IEnumerable<Type> GetTypesWithAttribute(Type attributeType) {
   foreach (Assembly assembly in AppDomain.CurrentDomain.GetAssemblies()) {
      // The attribute need to be tagged with the OptimizerPluginsAttribute.
      if (Attribute.GetCustomAttribute(assembly, typeof (OptimizerPluginsAttribute)) == null)
         continue;

      foreach (Type type in assembly.GetTypes()) {
         if (Attribute.GetCustomAttribute(type, attributeType, true) != null)
            yield return type;
      }
   }
}
```

Listing B.2: Rule binder initialization.

```
// Rewrite the goal using share equivalent class representatives
public BitSet ShareEquivalentGoal(BitSet goal) {
   // Copy the goal to get a working copy
   BitSet sharedGoal = (BitSet)goal.Clone();
   // shareEquivalenceMap contains all mappings BitSet -> Share Equivalent Representative BitSet
   foreach (Pair<BitSet, BitSet> mapper in shareEquivalenceMap)
      // If all of entire mapper.First is in sharedGoal...
      if ((sharedGoal & mapper.First) == mapper.First)
         // Then remove the ”from” bits and add in the ”to” bits, i.e. the share equivalent representatives
         sharedGoal.Subtract(mapper.First).Or(mapper.Second);
   return sharedGoal;
}

// Try to rewrite the goal using share equivalent class representatives and return true
// and the rewritten goal if successful, otherwise false.
public bool ShareEquivalentGoal(BitSet goal, out BitSet sharedGoal) {
   sharedGoal = ShareEquivalentGoal(goal);

   // If the new goal is different from the old and does not overlap the old except for any already
   // existing representatives in it, we have a successful rewrite.
   return sharedGoal != goal && !sharedGoal.Overlaps(goal - shareEquivalenceRepresentatives);
}
```

Listing B.3: Share Equivalence Rewrite algorithm.

B.3 Share Equivalence Rewrite

We wanted to include the full algorithm for rewriting a goal to a share equivalent goal. This is performed using `shareEquivalenceMap`, which maps share equivalent properties to their representatives. *ShareEquivalentGoal(BitSet goal)* rewrites a goal, while *ShareEquivalentGoal(BitSet goal, out BitSet sharedGoal)* rewrites and verifies that the new goal is a complete rewrite.

B.4 Complete, Unsimplified GeneratePlans

In Section 3.6.5 we introduced a simplified version of *GeneratePlans*. We also wanted to include the full, unsimplified version of it to show that the removed code is mostly debug output and housekeeping. The calls to DiagPrinter output the progress of a goal search, to be able to debug it. Lines like `rulesInvoked++;` update statistics variables.

```
/// <summary>
/// Generates plans satisfying the given goal. Multiple plans can be returned
/// if they are not comparable in cost or have different orderings.
/// </summary>
/// <param name=”goal”>The goal to fulfill.</param>
/// <param name=”limit”>Current cost limit for pruning (abort search if passing this).</param>
/// <returns>A PlanSet containing the best plans found.</returns>
public PlanSet GeneratePlans(BitSet goal, ICost limit) {
   DiagPrinter.Instance.DiagIndent(”search”);
   PlanSet plans;
   // If we already have a plan that satisfies the goal, return it.
   if (plansCache.TryGetValue(goal, out plans)) {
      plansFromCache++;
      if (plans == null)
         DiagPrinter.Instance.DiagWriteLine(”search”, ”UNREACHABLE␣from␣cache:␣{0}”, goal);
```

```
16          else
17             DiagPrinter.Instance.DiagWriteLine("search", "CACHE␣return:␣{0}", goal);
18
19          DiagPrinter.Instance.DiagUnindent("search");
20          return plans;
21       }
22
23       // Check if we can reach this goal with the current rule set.
24       if (GoalIsUnreachable(goal)) {
25          DiagPrinter.Instance.DiagWriteLine("search", "UNREACHABLE:␣{0}", goal);
26          DiagPrinter.Instance.DiagUnindent("search");
27
28          // Memoize the fact that it is unreachable, to avoid unnecessary
29          // GoalIsUnreachable()-calls that are O(n) for n rules
30          plansCache[goal] = null;
31          unreachablePlans++;
32          return null;
33       }
34
35       // We have a plan we have not made before, and we are able to create it!
36       plans = new PlanSet() {Properties = goal};
37
38       // If we've got a disjunct goal, try to get plans for this one also.
39       BitSet shared;
40       if (EnableDAG && ShareEquivalentGoal(goal, out shared)) {
41          DiagPrinter.Instance.DiagWriteLine("search", "Rewrote␣{0}␣to␣{1},␣firing␣off␣share␣
               equivalent␣search.",
42                                          goal, shared);
43
44          PlanSet sharingPlans = GeneratePlans(shared, BasicCost.Max);
45          if (sharingPlans != null) {
46             // Consider the plans where we share sub-plans.
47             plans.State = sharingPlans.State;
48             foreach (Plan plan in sharingPlans)
49                plans.AddPlan(plan);
50          }
51       }
52
53       // Use the search rule to search for plans.
54       foreach (ISearchRule searchRule in searchRules) {
55          if (! searchRule.IsRelevantTo(goal))
56             // The rule cannot satisfy the goal.
57             continue;
58
59          rulesInvoked++;
60          searchRule.Search(plans, limit);
61       }
62
63       if (plans.Count == 0)
64          plans = null;
65
66       plansCache[goal] = plans;
67       plansGenerated++;
68       DiagPrinter.Instance.DiagUnindent("search");
69       return plans;
70    }
```

Listing B.4: Complete, Unsimplified GeneratePlans.

B.5 BitSet

BitSet is the implementation of property sets using bit masks as storage. This allows for very compact storage and set operation like union and intersection becomes very fast since they can operate on the whole bit mask as a unit. We therefore include some selected code snippets to show how it is implemented.

B.5.1 BitSet Implementation

Listing B.5 shows how the data is stored inside the BitSet. The bit masks are stored as an array of ints, *data*, each element having room for 32 properties (ints are 4 bytes = 32 bits). A separate member *length* stores the number of bits in the bit mask that are in use.

```
1  /// <summary>The bits are internally stored as ints. 1 int up to 32, 2 up to 64 etc.</summary>
2  private int[] data;
3  /// <summary>Current number of bits stored.</summary>
4  private int length;
```

Listing B.5: BitSet private data.

Listing B.6 shows the interface to add and remove properties from the set. It closely resembles any other set implementation. Adding an element X will look up X's index in the central BitSetManager class and the set the corresponding bit to true.

```
1  public void Add(string property) {
2     this[manager[property]] = true;
3  }
4  public void Remove(string property) {
5     this[manager[property]] = false;
6  }
7  public bool Contains(string property) {
8     return this[manager[property]];
9  }
```

Listing B.6: BitSet single property interface.

Listing B.7 shows how the [] operator used in the previous listing is implemented. First, the correct array index is found by computing `index / 32` (each entry has room for 32 properties). The offset within the item is found as `index mod 32`. If reading the value, the binary value 1 is bit shifted *offset* positions to the left and bitwise intersection (AND) is computed with the stored bit mask. If the result is different from 0, the property is set. The procedure is similar for setting properties, but now the bit shifted value is intersected or unioned *into* the stored bit mask.

```
1  private bool this[int index] {
2     get {
3        // Get the correct array item, AND with 1 bitshifted
4        // to the correct position and return if it is not 0.
5        return ((this.data[index / 32] & (((int)1) << (index % 32))) != 0);
6     }
7     set {
8        if (value)
9           // OR with 1 bitshifted to the correct position.
10          this.data[index / 32] |= ((int)1) << (index % 32);
11       else
12          // AND with NOT (1 bitshifted to the correct position).
13          this.data[index / 32] &= ~(((int)1) << (index % 32));
14    }
```

```
15 }
```

Listing B.7: BitSet internal property implementation.

To show how efficient whole set operations on BitSets are, we include two of the available set operators that have been overloaded. The first one is set *intersection* between two BitSets. Intersection is performed by computing bitwise AND between the stored bit masks in both BitSets, returning a new BitSet with the result. Note that the number of AND operations carried out is *property count* / 32.

```
1 public static BitSet operator &(BitSet a, BitSet b) {
2    // Intersect all the data items (&) and construct a new BitSet.
3    int length = (a.length + 31) / 32;
4    int[] data = new int[length];
5    for (int i = 0; i < length; i++)
6       data[i] = a.data[i] & b.data[i];
7
8    return new BitSet(a.manager, data, a.length);
9 }
```

Listing B.8: BitSet set intersection operator.

The second set operator we include is *subset*, that is, if BitSet a is a subset of BitSet b. The subset operator $A \subseteq B$ can be expressed as $\left(A \cap \overline{B}\right) = \emptyset$. This is implemented by computing a AND !b for the stored bit masks in both sets and returning false if any of the results are non-zero.

```
1 public static bool operator <=(BitSet a, BitSet b) {
2    // Subset (A <= B) is implemented as (A & !B) == EMPTY.
3    int length = (a.length + 31) / 32;
4    for (int i = 0; i < length; i++)
5       if ((a.data[i] & ~b.data[i]) > 0)
6          return false;
7
8    return true;
9 }
```

Listing B.9: BitSet IsSubSet operator.

B.5.2 BitSet Minimization

The following code shows the BitSet minimization algorithm. It merges all properties that are always produced together. For example, if all properties known in the system are $\{a_1, a_2, b_1, c_1\}$ and a_1, a_2 is always produced together, the result will be $\{\{a_1, a_2\}, b_1, c_1\}$.

This can also be extended to prune properties that are never required or never produced.

```
1 /// <summary>Prepares the BitSetManager for use. This must be called before any bitsets can be
     used.
2 /// It will minimize the bit set properties and register all the mappings.</summary>
3 public void Prepare() {
4    ValidatePrepared(false);
5    prepared = true;
6    Dictionary<string, string> mappings = MinimizeProperties();
7    RegisterMappings(mappings);
8 }
9
10 /// <summary>Minimize the properties. Currently, this includes merging all properties
11 /// that are always produced together.</summary>
12 /// <returns>A dictionary that maps property name -> property name.
```

```
13 /// If for instance A and B are always produced together, it will contain {A->A, B->A}.</returns
       >
14 private Dictionary<string, string> MinimizeProperties() {
15    Dictionary<string, string> mappings = new Dictionary<string, string>();
16    // Register all direct mappings in all produced sets by default.
17    foreach (NestableHashSet<string> producedSet in produced)
18       foreach (string property in producedSet)
19          mappings[property] = property;
20
21    HashSet<string> allProducedProperties = new HashSet<string>();
22    // Get all the produced properties unioned together.
23    foreach (NestableHashSet<string> producedSet in produced)
24       allProducedProperties.UnionWith(producedSet);
25
26    // Now, foreach over all properties, visiting each combination {propA, propB} once,
27    // where propA < propB.
28    foreach (string propA in allProducedProperties) {
29       foreach (string propB in allProducedProperties) {
30          if (propA.CompareTo(propB) < 0) {
31             // Foreach over all produced sets. If a set contains propA or propB, add it
32             // to our set of sets.
33             HashSet<NestableHashSet<string>> setsContainingA = new HashSet<NestableHashSet<
                   string>>();
34             HashSet<NestableHashSet<string>> setsContainingB = new HashSet<NestableHashSet<
                   string>>();
35             foreach (NestableHashSet<string> p in produced) {
36                if (p.Contains(propA))
37                   setsContainingA.Add(p);
38                if (p.Contains(propB))
39                   setsContainingB.Add(p);
40             }
41             // Now setsContainingA contains all sets containing A and setsContainingB
42             // contains all sets containing B.
43             // If these two sets are set equals (contain the same elements), it means
44             // that propA and propB are always produced together.
45             if (setsContainingA.SetEquals(setsContainingB)) {
46                // If so, merge the properties by making propB map to whatever propA
47                // maps to. The reason we're mapping propB to mappings[propA] and
48                // not to propA, is that propA could have been remapped earlier itself.
49                mappings[propB] = mappings[propA];
50             }
51          }
52       }
53    }
54    return mappings;
55 }
56
57 /// <summary>Registers the minimized mappings. This means adding entries
58 /// to the nameToIndex and indexToName dictionaries.</summary>
59 /// <param name="mappings">The mappings to register.</param>
60 private void RegisterMappings(Dictionary<string, string> mappings) {
61    // mappings.Values contains all possible bit property variations.
62    // Register a physical property for all these.
63    foreach (string property in mappings.Values) {
64       // Only add each possible variation once.
65       if (!nameToIndex.ContainsKey(property)) {
66          nameToIndex[property] = indexToName.Count;
67          indexToName.Add(new HashSet<string>() { property });
68       }
```

```
69     }
70
71     // Then, add all the remappings (i.e. {B->A}).
72     foreach (KeyValuePair<string, string> pair in mappings) {
73        // Get the target index
74        int index = nameToIndex[pair.Value];
75        // Make the from-property point to the correct index
76        nameToIndex[pair.Key] = index;
77        // Add the from-property to the reverse lookup table
78        indexToName[index].Add(pair.Key);
79     }
80  }
```

Listing B.10: BitSet Minimization

B.6 Optimizer Tests

To verify that the optimizer implementation is working as expected, we have implemented several automated tests. The tests create a query to be optimized programmatically and then invoke the optimizer. Afterwards, they verify that something bad did not happen (e.g. Exception) or that the resulting query is the optimal one.

The test shown in Listing B.11 constructs the query in [Moe06, p. 38] and is the one used to generate the figures in Section 7.4.2. It feeds the optimizer the worst possible plan and asks the optimizer to optimize it using bushy enumeration. Afterwards it verifies that the resulting plan has the expected cost, and that it contains the correct nodes in the correct locations. Finally it asks the optimizer to optimize the same query using left-deep enumeration, and verifies that this plan has a worse cost estimate and is correctly laid out.

```
1   [Test]
2   public void SimpleJoins() {
3      double[] cardinalities = new double[] {1000, 2000, 2000, 1000};
4      double[] selectivities = new[] {0.005, 0.0001, 0.0001};
5      InitializeIndexStats(cardinalities);
6
7      // Now test bushy enumeration. This should yield the plan (0 X 1) X (2 X 3)
8      // with cost 1678.
9      TestNode[] expectedLookups = new[] {
10                                  GetLookup("word", "Mock0"),
11                                  GetLookup("word", "Mock1"),
12                                  GetLookup("word", "Mock2"),
13                                  GetLookup("word", "Mock3")
14                              };
15     TestNode[] expectedJoins = new TestNode[3];
16     expectedJoins[0] = GetMergeJoin(null, null, expectedLookups[1], expectedLookups[0]);
17     expectedJoins[1] = GetMergeJoin(null, null, expectedLookups[2], expectedLookups[3]);
18     expectedJoins[2] = GetMergeJoin(null, null, expectedJoins[0], expectedJoins[1]);
19     TestNode expected = GetMap(false, null, null, expectedJoins[2]);
20     expected.Properties = null;
21     expected = GetOutput(expected);
22     expectedJoins[2].CustomValidator = (n => Assert.AreEqual(new BasicCost(1678), n.Plan.Costs));
23
24     queryOptimization.JoinEnumeration = JoinEnumeration.Bushy;
25     OperatorNode result = queryOptimization.Optimize(GetSimpleJoinsInput(cardinalities,
           selectivities));
26     expected.AreEqualRecursively(result);
27
28     // Now test left-deep enumeration. This should yield the plan ((0 X 1) X 2) X 3
```

```
29    // with cost 1696.
30    expectedLookups = new[] {
31                        GetLookup("word", "Mock0"),
32                        GetLookup("word", "Mock1"),
33                        GetLookup("word", "Mock2"),
34                        GetLookup("word", "Mock3")
35                     };
36    expectedJoins[0] = GetMergeJoin(null, null, expectedLookups[1], expectedLookups[0]);
37    expectedJoins[1] = GetMergeJoin(null, null, expectedJoins[0], expectedLookups[2]);
38    expectedJoins[2] = GetMergeJoin(null, null, expectedJoins[1], expectedLookups[3]);
39    expected = GetMap(false, null, null, expectedJoins[2]);
40    expected.Properties = null;
41    expected = GetOutput(expected);
42    expectedJoins[2].CustomValidator = (n => Assert.AreEqual(new BasicCost(1696), n.Plan.Costs));
43
44    queryOptimization.JoinEnumeration = JoinEnumeration.LeftDeep;
45    result = queryOptimization.Optimize(GetSimpleJoinsInput(cardinalities, selectivities));
46    expected.AreEqualRecursively(result);
47 }
```

Listing B.11: SimpleJoins automated test.

B.7 AbstractTraverser

AbstractTraverser is the graph pattern matcher "work horse", as most of the time we are interested in evaluating a node with respect to other nodes in the graph — which are not necessarily directly adjacent.

It uses up to three sub-patterns, but is usable with less. A root matcher decides where traversal will begin. For every node matched by the root matcher, the concrete implementation's *IteratorFor()* is invoked to start iterating. For example, DescendantTraverser returns an enumerator for all the descendants of the node, whereas ChildMatcher only returns the immediate children. The root matcher is optional — if it is not specified, the Node given as input to the traverser's *Search()*-method is used as the only root.

If a `StopPattern` is specified, matching stops as soon as a match is found. If no match can be found, the traverser will declare that is has not matched.

If a `TakeWhile`-pattern is specified, matching will continue as long as that pattern matches. If no stop pattern has been specified, the traverser returns whether its `TakeWhile` matches. If both are specified, both will need to match for the traverser to declare that it has matched.

The following pattern, used in the OrderingExtracter-pre-processor illustrates both concepts:

```
1 Match.NodeWithBehaviour(OperatorBehaviour.OrderPreserving)
2             .WithDescendant(Match.NodeWithBehaviour(OperatorBehaviour.OrderPreserving),
3                          Match.NodeOfType("SortOperator").GroupAs("sort"));
```

It matches any operator that preserves order, which has a Sort-operator as a descendant with only order preserving operators in between.

```
1 /// <summary>
2 /// Invoked for every root found. Steers the direction of the traversal.
3 /// </summary>
4 public abstract IEnumerable<Node> IteratorFor(Node node);
5
6 public override bool Search(Node node) {
7    Debug.Assert(!(TakeWhile == null && StopPattern == null), "TakeWhile␣and␣StopPattern␣cannot␣
        both␣be␣null");
8
```

```
9       // First, determine roots we'll find iterators for.
10      HashSet<Node> roots;
11      if (RootMatcher != null) {
12         if (!RootMatcher.Search(node))
13            return false;
14         roots = new HashSet<Node>(RootMatcher.MatchedNodes);
15      }
16      else
17      {
18         roots = new HashSet<Node>() {node};
19      }
20
21      // Iterate every root.
22      foreach (Node root in roots) {
23         foreach (Node descendant in IteratorFor(root)) {
24            // If we have a stop pattern and it matches, we're done.
25            if (StopPattern != null && StopPattern.Search(descendant)) {
26               return HasMatched = true;
27            }
28            // If not, continue if we can.
29            if (! (TakeWhile == null || TakeWhile.Search(descendant)))
30               break;
31         }
32      }
33
34      return HasMatched = (StopPattern == null || StopPattern.HasMatched) && (TakeWhile == null ||
            TakeWhile.HasMatched);
35   }
```

Listing B.12: Important parts of AbstractTraverser.

B.8 LogicalJoinTransformer

This is the full implementation of the LogicalJoinTransformer. See Section 5.2.1 for an explanation.

```
1    [Preprocessor]
2    public class LogicalJoinTransformer : AbstractTransformationRule {
3       public override bool Iterative {
4          get { return false; }
5       }
6
7       public override IEnumerable<Type> DependsOn {
8          get { yield break; }
9       }
10
11      public override AbstractNodeMatcher Pattern {
12         get {
13            // Match HybridHashJoins and MergeJoins.
14            return (Match.NodeOfType("HybridHashJoinOperator") | Match.NodeOfType("
                 MergeJoinOperator")).GroupAs("join");
15         }
16      }
17
18      public override bool Fire(GraphSearcher graphSearcher) {
19         List<Node> joins = graphSearcher.Groups["join"];
20         foreach (OperatorNode join in joins) {
21            IList<string> joinKey = new List<string>();
22
```

```
23          // To convert into a logical join, we need to determine the proper join keys.
24          // For simplicity, we've currently limited ourself to having non-composite keys.
25          if (join.OperatorType == typeof (HybridHashJoinOperator)) {
26             // Keys are a list of list of attributes.
27             IList<IList<string>> joinKeys = (IList<IList<string>>) join["JoinKeys"];
28             if (joinKeys.Count != 1)
29                throw new OptimizerException("Only␣single␣join␣keys␣are␣accepted␣in␣the␣
                      prototype.");
30             joinKey = joinKeys.Single();
31          }
32          else if (join.OperatorType == typeof (MergeJoinOperator)) {
33             // Keys are determined by the #joinPrefix first attributes.
34             int joinPrefix = (int) join["JoinPrefix"];
35             if (joinPrefix != 1)
36                throw new OptimizerException("Only␣single␣join␣keys␣are␣accepted␣in␣the␣
                      prototype.");
37
38             for (int i = 0; i < join.Children.Count; i++)
39                joinKey.Add(((OperatorNode) join.Children[i]).OutputTypeDescriptor.RecordType[0].
                      Name);
40          }
41
42          // Keys properly determined, set them and change the operator type.
43          join["JoinKeys"] = joinKey;
44          join.OperatorType = typeof (JoinOperator);
45       }
46       return true;
47    }
48 }
```

Listing B.13: LogicalJoinTransformer implementation.

B.9 Lookup Rule

To show how a rule is actually implemented in full, we have chosen to include the full implementation of two rules, one base rule and one search rule, the first being LookupRule as this is the simplest rule.

LookupRule is a base rule (no inputs), and therefore inherits AbstractBaseRule, which only implements a few properties from IBaseRule.

We have also included its rule binder, LookupRuleBinder, which is responsible for instantiating the rule. It inherits from AbstractRuleBinder, which implements functionality needed for most rule binders.

We have included all the base classes for completeness. We have removed some API documentation to save space.

```
1  /// <summary>Base rule representing MARS' Lookup operator.</summary>
2  public class LookupRule : AbstractBaseRule{
3     public LookupRule(QueryOptimization queryOptimization)
4        : base(queryOptimization)
5     { }
6
7     /// <summary>Statistics for this index lookup.</summary>
8     public BasicPlanSetState Stats { get; set; }
9     /// <summary>Name of the index to look up into.</summary>
10    public string Index { get; set; }
11    /// <summary>Word to look up.</summary>
12    public string Word { get; set; }
```

```
13     /// <summary>The order produced by this lookup (if it us clustered).</summary>
14     public OrderProxy Order { get; set; }
15
16     /// <summary>
17     /// Initialize a plan set with properties from the rule, like tuple size, cardinality etc.
18     /// </summary>
19     /// <param name="plans">The plan set to initialize.</param>
20     public override void Initialize(PlanSet plans) {
21        plans.State = Stats;
22     }
23
24     /// <summary>
25     /// Updates the given plan with information from this rule, like
26     /// costs, order and so on.
27     /// </summary>
28     /// <param name="plan">The plan (constructed by this rule).</param>
29     public override void UpdatePlan(Plan plan) {
30        base.UpdatePlan(plan);
31
32        // Initialize to the ordering provided by the index.
33        plan.OrderingState = new OrderingState(queryOptimization.OrderManager, Order.Order);
34
35        // Set sharing for the plan if this rule is the equivalence class representative.
36        plan.Sharing = queryOptimization.BitSetManager.Empty;
37        if (queryOptimization.ShareEquivalenceClasses.ContainsKey(this)) {
38           plan.Sharing.Or(
39              queryOptimization.BitSetManager.GetWithValues(new NodeAttribute(Nodes.Single(), null
                    ).PropertyName));
40           plan.Shared = true;
41        }
42     }
43
44     /// <summary>Builds the physical algebra node for the given plan.</summary>
45     /// <param name="plan"></param>
46     /// <returns>The physical algebra node</returns>
47     public override OperatorNode BuildAlgebra(Plan plan) {
48        OperatorNode node;
49        if (GetOrCreatePhysicalNode(plan, typeof (LookupOperator), out node)) {
50           node.Properties = Nodes.Single().Properties;
51           node.Rules = Nodes.Single().Rules;
52        }
53        return node;
54     }
55
56     /// <summary>
57     /// Determines if this and the other rule is structurally identical, i.e. if they have
58     /// the same configured properties, the same join keys etc.
59     /// </summary>
60     /// <param name="other">Rule to compare with.</param>
61     /// <returns>If the rules are structurally identical.</returns>
62     public override bool StructurallyIdentical(IProducerRule other) {
63        // Lookups are structurally identical if they look up the same word in the same index.
64        LookupRule otherLookup = other as LookupRule;
65        return otherLookup != null && Index.Equals(otherLookup.Index) && Word.Equals(otherLookup.
              Word);
66     }
67
68     /// <summary>
69     /// This method will be called if this rule is determined to be share equivalent with another
```

```
        rule.
70     /// It should map any attributes the other rule produces to be share equivalent to the
           corresponding attributes
71     /// produced by this rule.
72     /// </summary>
73     /// <param name="from">The rule to map attributes from.</param>
74     /// <param name="equivalenceClassAttributeMappings">The dictionary the mappings should be
           added to.</param>
75     public override void MapShareEquivalentAttributes(IProducerRule from,
76                                                       Dictionary<NodeAttribute, NodeAttribute>
77                                                           equivalenceClassAttributeMappings) {
78        // Map all output attributes.
79        foreach (Field field in Nodes.Single().RecordDescriptor)
80           equivalenceClassAttributeMappings.Add(from.Nodes.Single().RecordDescriptor[field.Name].
                 NodeAttribute,
81                                                  field.NodeAttribute);
82     }
83
84     public override string ToString() {
85        return string.Format("LookupRule(Index:{0},␣Word:{1},␣Produces:␣{2}", Index, Word,
             Produced.BitSet);
86     }
87
88     /// <summary>
89     /// The cost for only the operator represented by this rule, applied
90     /// on the top of the given plan.
91     /// </summary>
92     /// <param name="plan">Plan with this rule on the top.</param>
93     /// <returns>The cost of this exact rule.</returns>
94     protected override BasicCost Cost(Plan plan) {
95        double cost = BasicCost.ReadSequential(BasicCost.PageAccesses(Stats.ResultSetSize));
96        return new BasicCost(cost, cost);
97     }
98  }
```

Listing B.14: LookupRule implementation.

```
1   /// <summary>Rule binder for Lookup operators.</summary>
2   [RuleBinder] public class LookupRuleBinder :
3   AbstractLeafRuleBinder<LookupRule>, IProducesMapper, IFieldMapper {
4      /// <summary>The pattern in the input query this rule binder will instantiate rules for.</
           summary>
5      public override AbstractNodeMatcher Pattern {
6         get { return Match.NodeOfType("LookupOperator"); }
7      }
8
9      /// <summary>The types of operators the implementer is a mapper for.</summary>
10     public override Type MapperForType {
11        get { return typeof (LookupOperator); }
12     }
13
14     #region IFieldMapper Members
15
16     // Returns all the output fields for the given lookup node.
17     private IEnumerable<Field> FieldsFor(QueryOptimization queryOptimization, OperatorNode node)
           {
18        return queryOptimization.SystemMetadata.GetIndexSchema((string) node["IndexName"], (string
             ) node["Word"]);
19     }
20
```

```
21      /// <summary>
22      /// This method is called on the corresponding mapper for all nodes in the query in
23      /// topological order and should set field names for all NodeAttributes in the nodes
24      /// output using OperatorNode’s RecordDescriptor property. It should also convert
25      /// all NodeAttribute references to field names.
26      /// </summary>
27      /// <param name=”queryOptimization”>The current instance of QueryOptimization.</param>
28      /// <param name=”node”>The node to set field names for.</param>
29      public void SetFields(QueryOptimization queryOptimization, OperatorNode node) {
30         //Add all output attributes.
31         foreach (Field field in FieldsFor(queryOptimization, node))
32            node.RecordDescriptor.AddFieldForAttr(
33               new Field(new NodeAttribute(node, field.NodeAttribute.AttributeName), field.Name,
                     field.Type));
34      }
35
36      #endregion
37
38      #region IProducesMapper Members
39
40      /// <summary>
41      /// This method is called on the corresponding mapper for all nodes in the query in
42      /// topological order and should set the produced attributes
43      /// using OperatorNode’s RecordDescriptor property. It should also convert
44      /// all field name references to NodeAttribute.
45      /// </summary>
46      /// <param name=”queryOptimization”>The current instance of QueryOptimization.</param>
47      /// <param name=”node”>The node to set produced attributes for.</param>
48      public virtual void SetProduces(QueryOptimization queryOptimization, OperatorNode node) {
49         //Add all output attributes.
50         foreach (Field field in FieldsFor(queryOptimization, node))
51            node.RecordDescriptor.AddFieldForName(
52               new Field(new NodeAttribute(node, field.NodeAttribute.AttributeName), field.Name,
                     field.Type));
53      }
54
55      #endregion
56
57      /// <summary>
58      /// This method is called by the optimizer to get the rules this rule binder produces.
59      /// </summary>
60      /// <param name=”queryOptimization”>The currrent QueryOptimization instance.</param>
61      /// <param name=”matches”>IEnumerable of matches to the pattern declared by the rule binder
            .</param>
62      /// <returns>IEnumerable of rules that should be added to the search by the optimizer.</
            returns>
63      public override IEnumerable<IProducerRule> GetRules(QueryOptimization queryOptimization,
64                                                         IEnumerable<OperatorNode> nodes) {
65         List<IProducerRule> rules = new List<IProducerRule>();
66
67         // For each node, instantiate a rule and set properties.
68         foreach (OperatorNode node in nodes) {
69            // Instantiate rule
70            LookupRule rule = CreateRule(queryOptimization, node);
71            rules.Add(rule);
72
73            // Index and lookup term.
74            rule.Index = (string) rule.Nodes.Single()[”IndexName”];
75            rule.Word = (string) rule.Nodes.Single()[”Word”];
```

```
76
77          // Produced attributes.
78          NestableHashSet<string> produced = new NestableHashSet<string>
79                                              {new NodeAttribute(rule.Nodes.Single(), null).
                                                     PropertyName};
80          foreach (Field field in node.RecordDescriptor)
81             produced.Add(field.NodeAttribute.PropertyName);
82          rule.Produced = queryOptimization.BitSetManager.GetBitSet(PropertyUsage.Produced,
                 produced);
83
84          // Orderings.
85          OrderDescription orderDesc = new OrderDescription(
86             queryOptimization.SystemMetadata.GetIndexOrder(rule.Index)
87                .Select(
88                item =>
89                new OrderDescriptionItem(new NodeAttribute(node, item.Property).PropertyName,
                       item.Ascending)));
90          rule.Order = queryOptimization.OrderManager.GetOrder(orderDesc, true);
91          rule.Stats = queryOptimization.SystemMetadata.GetIndexStats(rule.Index, rule.Word);
92       }
93       return rules;
94    }
95 }
```

Listing B.15: LookupRuleBinder implementation.

```
1  /// <summary>Base class for base rules.</summary>
2  public abstract class AbstractBaseRule : AbstractRule, IBaseRule {
3     public AbstractBaseRule(QueryOptimization queryOptimization)
4        : base(queryOptimization)
5     { }
6
7     #region IBaseRule Members
8
9     /// <summary>Produced properties for this rule.</summary>
10    public BitSetProxy Produced { get; set; }
11
12    /// <summary>Filter for this rule, equal to produced properties.</summary>
13    public BitSet Filter {
14       get { return Produced.BitSet; }
15    }
16
17    /// <summary>Id of this rule.</summary>
18    public int Id { get; set; }
19
20    public abstract bool StructurallyIdentical(IProducerRule other);
21
22    public abstract void MapShareEquivalentAttributes(IProducerRule from,
23                                               Dictionary<NodeAttribute, NodeAttribute>
24                                                  EquivalenceClassAttributeMappings);
25    public abstract void Initialize(PlanSet plans);
26
27    #endregion
28 }
```

Listing B.16: AbstractBaseRule implementation.

```
1  /// <summary>Abstract base class for all rules.</summary>
2  public abstract class AbstractRule : IRule {
3     protected QueryOptimization queryOptimization;
```

```
4
5       public AbstractRule(QueryOptimization queryOptimization) {
6          this.queryOptimization = queryOptimization;
7       }
8
9       /// <summary>Nodes this rule was instantiated from.</summary>
10      public IList<OperatorNode> Nodes { get; set; }
11
12      // IRule
13
14      #region IRule Members
15
16      /// <summary>
17      /// Updates the properties of the plan (costs, ordering, sharing).
18      /// Will calculate costs for both DAGs and trees.
19      /// </summary>
20      /// <param name="plan">The plan to update.</param>
21      public virtual void UpdatePlan(Plan plan) {
22         plan.Rule = this;
23         UpdateCosts(plan);
24      }
25
26      /// <summary>
27      /// Updates the costs for the given plan, automatically choosing between DAG and tree
             calculation
28      /// algorithms.
29      /// </summary>
30      /// <param name="plan">The plan to calculate costs for.</param>
31      public virtual void UpdateCosts(Plan plan) {
32         // Get the cost for this operator.
33         BasicCost cost = Cost(plan);
34         // If only one child, just sum them.
35         if (plan.Children.Count == 1)
36            cost += (BasicCost) plan.OnlyChild.Costs;
37         // If more than one child, we may need to do DAG costing.
38         else if (plan.Children.Count > 1)
39            cost += BasicCost.InputCosts(plan.Children, null);
40         plan.Costs = cost;
41      }
42
43      public abstract OperatorNode BuildAlgebra(Plan plan);
44
45      /// <summary>
46      /// Return the cost for the subtree with this rule on the top,
47      /// using the cost algorithms for DAGs.
48      /// </summary>
49      /// <param name="plan">The plan (constructed by this rule).</param>
50      /// <param name="reads">How many reads of the plan.</param>
51      /// <returns>The costs for the subtree.</returns>
52      public virtual ICost DagCosts(Plan plan, int reads) {
53         BasicCost planCosts = (BasicCost) plan.Costs;
54         // If we've already been executed this many times, the reads are free.
55         if (reads < planCosts.Passes)
56            return BasicCost.Zero;
57
58         // Our cost:
59         int additional = reads - planCosts.Passes;
60         planCosts.Passes = reads;
61         plan.Costs = planCosts; // BasicCost is a struct, so set the entire thing.
```

```
62          BasicCost result = Cost(plan)*additional;
63
64          // Cost of inputs:
65          foreach (Plan subPlan in plan.Children)
66             result += (BasicCost) subPlan.Rule.DagCosts(subPlan, reads);
67
68          return result;
69       }
70
71       #endregion
72
73       /// <summary>
74       /// Sets the node out-parameter to the operator root-node for the plan.
75       /// Returns true if the node was created and false if it was found in the reconstruction
              table.
76       /// </summary>
77       protected bool GetOrCreatePhysicalNode(Plan plan, Type operatorType, out OperatorNode node) {
78          if (queryOptimization.ReconstructionTable.TryGetValue(plan, out node))
79             return false;
80
81          node = createPhysicalNode(plan, operatorType);
82          return true;
83       }
84
85       // Creates the physical operator node for the given plan and operator type.
86       private OperatorNode createPhysicalNode(Plan plan, Type operatorType) {
87          // Root node for the plan:
88          OperatorNode node = new OperatorNode(operatorType);
89          node.ByRule = GetType();
90          if (Nodes != null && Nodes.Count > 0)
91             node.Id = Nodes.First().Id;
92          node.Plan = plan;
93
94          // Also create children.
95          foreach (Plan subPlan in plan.Children)
96             node.AddChild(subPlan.Rule.BuildAlgebra(subPlan));
97
98          // Memoize it so new requests for the same plan gets the same node,
99          // important for creating DAGs.
100         queryOptimization.ReconstructionTable[plan] = node;
101
102         return node;
103      }
104
105      protected abstract BasicCost Cost(Plan plan);
106   }
```

Listing B.17: AbstractRule implementation.

```
1    /// <summary>Base class for rule binders, offering common functionality.</summary>
2    /// <typeparam name="RuleType">The type of rule to bind.</typeparam>
3    public abstract class AbstractRuleBinder<RuleType> : IRuleBinder where RuleType
4    : IProducerRule {
5       #region IRuleBinder Members
6
7       /// <summary>Node pattern to match in the operator graph during initialization.</summary>
8       public abstract AbstractNodeMatcher Pattern { get; }
9
10      /// <summary>
11      /// This method is called by the optimizer to get the rules this rule binder produces.
```

```
12      /// The default implementation instantiates a rule for each pattern match
13      /// and sets its produced properties to RULE_APPLIED.
14      /// </summary>
15      public virtual IEnumerable<IProducerRule> GetRules(QueryOptimization queryOptimization,
16                                                       IEnumerable<OperatorNode> nodes) {
17         List<IProducerRule> rules = new List<IProducerRule>();
18         // For each match, instantiate a rule and set properties.
19         foreach (OperatorNode node in nodes) {
20            RuleType rule = CreateRule(queryOptimization, node);
21            rule.Produced = queryOptimization.BitSetManager.GetBitSet(PropertyUsage.Produced,
22                                                                  new NodeAttribute(rule.Nodes.Single(), null
                                                                       ).
23                                                                    PropertyName);
24            rules.Add(rule);
25         }
26         return rules;
27      }
28
29      #endregion
30
31      /// <summary>
32      /// Creates a rule instance and sets common properties.
33      /// </summary>
34      /// <param name="queryOptimization">The query optimization instance.</param>
35      /// <param name="node">Node to get commoin properties from.</param>
36      /// <returns>The created rule instance.</returns>
37      protected RuleType CreateRule(QueryOptimization queryOptimization, OperatorNode node) {
38         RuleType rule = (RuleType) Activator.CreateInstance(typeof (RuleType), queryOptimization);
39         rule.Id = node.Id;
40         rule.Nodes = new List<OperatorNode> {node};
41         node.Rules = new List<IRule> {rule};
42         return rule;
43      }
44   }
```

Listing B.18: AbstractRuleBinder implementation.

B.10 Selection Rule

To show how a search rule is actually implemented, we have chosen to include most of the implementation of the simplest one, SelectionRule. This rule constructs selection operators.

SelectionRule is a unary rule (only one input), and therefore inherits UnaryRule. The latter implements the basic search strategy for unary rules: construct all plans where the rule itself is the topmost one. UnaryRule inherits from AbstractSearchRule which implements basic functionality required for all search rules.

We have also included its rule binder, SelectionRuleBinder.

```
1   /// <summary>This rule is reponsible for constructing selection operators in query plans.</
        summary>
2   public class SelectionRule : UnaryRule {
3      public SelectionRule(QueryOptimization queryOptimization)
4         : base(queryOptimization)
5      {}
6
7      /// <summary>Cost of evaluating the predicates for each record.</summary>
8      public double PredicateCost { get; set; }
9      /// <summary>Set of FDs induced by the selection.</summary>
10     public HashSet<Dependency> InducedDependencies { get; set; }
```

```
11     /// <summary>The selection predicate.</summary>
12     public ExpressionProperty SelectionFilter { get; set; }
13
14     /// <summary>
15     /// Updates the given plan with ordering, leaves the rest to UnaryRule.
16     /// </summary>
17     public override void UpdatePlan(Plan plan) {
18        base.UpdatePlan(plan);
19        plan.OrderingState = plan.OnlyChild.OrderingState.Apply(InducedDependencies);
20     }
21
22     public override OperatorNode BuildAlgebra(Plan plan) {
23        OperatorNode node;
24        if (GetOrCreatePhysicalNode(plan, typeof (SelectOperator), out node)) {
25           node.Properties = Nodes.First().Properties;
26           node.Rules = Nodes.First().Rules;
27        }
28        return node;
29     }
30
31     /// <summary>
32     /// The cost for the selection, which is predicate cost times cardinality.
33     /// </summary>
34     protected override BasicCost Cost(Plan plan) {
35        BasicPlanSetState childState = (BasicPlanSetState) plan.OnlyChild.PlanSet.State;
36        double cost = BasicCost.CpuCosts(childState.Cardinality, PredicateCost);
37        return new BasicCost(cost, cost);
38     }
39
40     /// <summary>
41     /// Determines if this and the other rule is structurally identical,
42     /// Selections are structurally identical if the filter predicates are equal.
43     /// </summary>
44     public override bool StructurallyIdentical(IProducerRule other) {
45        SelectionRule otherSelection = other as SelectionRule;
46        return otherSelection != null && SelectionFilter.Equals(otherSelection.SelectionFilter);
47     }
48  }
```

Listing B.19: SelectionRule implementation.

```
1   /// <summary>Rule binder for SelectionRule.</summary>
2   [RuleBinder] public class SelectionRuleBinder :
3   AbstractFilterRuleBinder<SelectionRule>, IDependencyMapper, IProducesMapper{
4      /// <summary>Node pattern to match in the operator graph during initialization, matching
          SelectOperator.</summary>
5      public override AbstractNodeMatcher Pattern {
6         get { return Match.NodeOfType("SelectOperator"); }
7      }
8
9      #region IDependencyMapper Members
10
11     /// <summary>The types of operator the implementes is a mapper for: SelectOperator.</summary>
12     public override IEnumerable<Type> MapperFor {
13        get { return new List<Type> {typeof (SelectOperator)}; }
14     }
15
16     /// <summary>Selection depends on all attributes accessed by the predicate.</summary>
17     public void SetDependencies(OperatorNode selectNode) {
18        // To properly set the dependencies, we'll invoke MARS' parser of the expression.
```

```
19
20          // Prepare some stuff needed for parsing.
21          IRecordTypeDescriptor inputFieldTypes = selectNode.RecordDescriptor.AsRecordTypeDescriptor
                ();
22          string expression = ((ExpressionProperty) selectNode["Filter"]).Expression;
23          Dictionary<string, object> symbolsMap = MakeSymbolsMap(expression, inputFieldTypes);
24
25          // Parse the expression, which gives us an easily traversable AST.
26          Expression linqExpression = ParseExpression(expression, symbolsMap);
27
28          // Visit the expression. The visitor keeps references to depended-on attributes.
29          FilterVisitor visitor = new FilterVisitor();
30          visitor.Initialize();
31          visitor.Visit(linqExpression);
32
33          // Get the dependencies from the visitor.
34          selectNode.Dependencies.Add(new HashSet<NodeAttribute>());
35          foreach (int visitedIndex in visitor.ReferencedIndexes)
36             selectNode.Dependencies[0].Add(selectNode.OnlyInput.RecordDescriptor[visitedIndex].
                   NodeAttribute);
37       }
38
39       #endregion
40
41       #region IProducesMapper Members
42
43       /// <summary>
44       /// Pass through the input attributes and replace references in the predicate with
             NodeAttributes.
45       /// </summary>
46       public override void SetProduces(QueryOptimization queryOptimization, OperatorNode node) {
47          base.SetProduces(queryOptimization, node);
48
49          string expression = ((ExpressionProperty) node["Filter"]).Expression;
50          foreach (Field field in node.OnlyInput.RecordDescriptor) {
51             System.Text.RegularExpressions.Regex r =
52                new System.Text.RegularExpressions.Regex("\\b" +
53                                                System.Text.RegularExpressions.Regex.Escape(
54                                                   field.NodeAttribute.AttributeName) + "\\b");
55             expression = r.Replace(expression, field.NodeAttribute.PropertyName);
56          }
57          node["Filter"] = new ExpressionProperty(expression);
58       }
59
60       #endregion
61
62       public override IEnumerable<IProducerRule> GetRules(QueryOptimization queryOptimization,
63                                                IEnumerable<OperatorNode> nodes) {
64          IEnumerable<IProducerRule> rules = base.GetRules(queryOptimization, nodes);
65          foreach (SelectionRule rule in rules) {
66             setSelectivity(rule);
67
68             // This is to be able to set selectivity and predicate cost in tests.
69             if (rule.Nodes.Single()["Selectivity"] != null)
70                rule.Selectivity = (double)rule.Nodes.Single()["Selectivity"];
71             if (rule.Nodes.Single()["PredicateCost"] != null)
72                rule.PredicateCost = (double)rule.Nodes.Single()["PredicateCost"];
73
74             setRuleDependencies(queryOptimization, rule);
```

```
75        }
76        return rules;
77    }
78
79    // Sets the selectivity for the rule.
80    private static void setSelectivity(SelectionRule rule) {
81        // We don't have anything to guesstimate selectivities from yet.
82        rule.Selectivity = 0.1;
83    }
84
85    // Sets dependecies for the selection rule by parsing the predicate expression.
86    private static void setRuleDependencies(QueryOptimization queryOptimization, SelectionRule
          rule) {
87        // Get the LINQ Expression for this selection
88        RecordDescriptor inputDescriptor = rule.Nodes.Single().RecordDescriptor;
89        rule.SelectionFilter = (ExpressionProperty) rule.Nodes.Single()["Filter"];
90        string expression = rule.SelectionFilter.Expression;
91        Dictionary<string, object> symbolsMap = MakeSymbolsMap(expression,
92                                                    inputDescriptor.AsRecordTypeDescriptor());
93        Expression linqExpression = ParseExpression(expression, symbolsMap);
94
95        // Visit the expression
96        FilterVisitor visitor = new FilterVisitor();
97        visitor.Initialize();
98        visitor.Visit(linqExpression);
99
100       // Add required properties found by the visitor
101       NestableHashSet<string> required = new NestableHashSet<string>();
102       foreach (int index in visitor.ReferencedIndexes)
103           required.Add(rule.Nodes.Single().RecordDescriptor[index].NodeAttribute.PropertyName);
104
105       rule.Required = new List<BitSetProxy>
106                   {queryOptimization.BitSetManager.GetBitSet(PropertyUsage.Required, required
                          )};
107
108       // Determine dependencies
109       rule.InducedDependencies = new HashSet<Dependency>();
110       foreach (Pair<object, object> equivalence in visitor.Equivalences) {
111           Dependency dependency = null;
112
113           // Both sides of the equivalence are columns in the data stream, add COL=COL dependency
114           if (equivalence.First is FieldIndex && equivalence.Second is FieldIndex)
115               dependency = queryOptimization.OrderManager.GetDependency(
116                   inputDescriptor[((FieldIndex) equivalence.First).Index].NodeAttribute.
                          PropertyName,
117                   inputDescriptor[((FieldIndex) equivalence.Second).Index].NodeAttribute.
                          PropertyName, true);
118
119           // The left operand is a constant, right one is a column. Add -->COL dependency
120           if (equivalence.First is Constant && equivalence.Second is FieldIndex)
121               dependency = queryOptimization.OrderManager.GetDependency(
122                   null,
123                   inputDescriptor[((FieldIndex) equivalence.Second).Index].NodeAttribute.
                          PropertyName, false);
124
125           // The right operand is a constant, left one is a column. Add -->COL dependency
126           if (equivalence.First is FieldIndex && equivalence.Second is Constant)
127               dependency = queryOptimization.OrderManager.GetDependency(
128                   null,
```

```
129                 inputDescriptor[((FieldIndex) equivalence.First).Index].NodeAttribute.
                        PropertyName, false);
130
131             if (dependency != null)
132                rule.InducedDependencies.Add(dependency);
133          }
134       }
135
136       /// <summary>
137       /// Passes through input field names and convert NodeAttribute refs back to field names.
138       /// </summary>
139       public override void SetFields(QueryOptimization queryOptimization, OperatorNode node) {
140          base.SetFields(queryOptimization, node);
141          string expression = ((ExpressionProperty) node["Filter"]).Expression;
142          foreach (Field field in node.OnlyInput.RecordDescriptor.GetFieldsWithMappings(
                 queryOptimization)) {
143             Regex.Regex r =
144                new Regex.Regex("\\b" + Regex.Regex.Escape(field.NodeAttribute.PropertyName) + "\\b"
                      );
145             expression = r.Replace(expression, field.Name);
146          }
147          node["Filter"] = new ExpressionProperty(expression);
148       }
149       ...
150    }
```

Listing B.20: SelectionRuleBinder implementation.

```
1  /// <summary>General base class for all unary rules (one input), offering
2  /// basic search functionality.</summary>
3  public abstract class UnaryRule : AbstractSearchRule {
4     public UnaryRule(QueryOptimization queryOptimization)
5        : base(queryOptimization)
6     { }
7
8     /// <summary>
9     /// The rquired bitset for the only input.
10    /// </summary>
11    public BitSet OnlyRequired {
12       get { return Required[0].BitSet; }
13    }
14
15    /// <summary>
16    /// Guides the search for this rule instance. Offers the basic search functionality
17    /// for unary rules: generating all possible plans with this rule on the top.
18    /// </summary>
19    /// <param name="plans">PlanSet to add plans to (also defines goal properties).</param>
20    /// <param name="limit">Abort the search if passing this cost limit (pruning).</param>
21    public override void Search(PlanSet plans, ICost limit) {
22       DiagPrinter.Instance.DiagWriteLine("search", "UnaryRule producing {0} searching for {1}."
              , Produced.BitSet,
23                                    (plans.Properties - Produced.BitSet) | Required[0].BitSet);
24       // Get the possible input plans, i.e. plans with needed properties,
25       // except the ones we produce ourselves.
26       PlanSet input = queryOptimization.GeneratePlans((plans.Properties - Produced.BitSet) |
              Required[0].BitSet,
27                                               limit);
28       if (input == null) {
29          DiagPrinter.Instance.DiagWriteLine("search", "0 plans.");
30          return; // No plans
```

```
31         }
32         DiagPrinter.Instance.DiagWriteLine(”search”, ”{0}␣plans.”, input.Count);
33
34         if (plans.Count == 0)
35            // First plan, so set some state.
36            plans.State = CalcPlanSetState((BasicPlanSetState) input.State);
37
38         // Add each input plan to the PlanSet, updating their properties.
39         foreach (Plan plan in input) {
40            Plan newPlan = new Plan {Children = new List<Plan> {plan}};
41            UpdatePlan(newPlan);
42            plans.AddPlan(newPlan);
43         }
44      }
45
46      /// <summary>
47      /// Calculates the PlansSetState for the output based on the input.
48      /// </summary>
49      /// <param name=”input”>Input PlanSetState.</param>
50      /// <returns>Output PlanSetState.</returns>
51      protected virtual BasicPlanSetState CalcPlanSetState(BasicPlanSetState input) {
52         return input.Filter(Selectivity);
53      }
54
55      public override void UpdatePlan(Plan plan) {
56         base.UpdatePlan(plan);
57
58         // Sharing is at least the sharing of the sub plan
59         plan.Sharing = (BitSet) plan.OnlyChild.Sharing.Clone();
60
61         // If this is not a representative or the child is not shared, no more sharing is possible
               .
62         if (!queryOptimization.ShareEquivalenceClasses.ContainsKey(this) || !plan.OnlyChild.Shared
               )
63            return;
64
65         // Add this rule to the sharing bitset and set this plan as shareable.
66         plan.Sharing.Or(
67            queryOptimization.BitSetManager.GetWithValues(new NodeAttribute(Nodes.Single(), null).
                  PropertyName));
68         plan.Shared = true;
69      }
70   }
```

Listing B.21: UnaryRule implementation.

```
1   /// <summary>Base class for search rules, implementing required members and Filter caching.</
        summary>
2   public abstract class AbstractSearchRule : AbstractRule, ISearchRule {
3      // We're using a bool on the side instead of BitSet? (nullable bitset) becase it is actually
           less overhead.
4      private bool cached = false;
5      private BitSet cachedFilter;
6
7      public AbstractSearchRule(QueryOptimization queryOptimization)
8         : base(queryOptimization) {
9         // Default selectivity as this may also be a non-limiting rule.
10        Selectivity = 1;
11     }
12
```

```
13    /// <summary>Selectivity for this rule. 1 for non-limiting rules.</summary>
14    public double Selectivity { get; set; }
15
16    #region ISearchRule Members
17
18    /// <summary>Properties produced by this rule.</summary>
19    public BitSetProxy Produced { get; set; }
20
21    /// <summary>List of properties required for this rule.</summary>
22    public IList<BitSetProxy> Required { get; set; }
23
24    /// <summary>
25    /// Automatically generates and caches the Filter property based
26    /// on the Produced and Required properties.
27    /// </summary>
28    public virtual BitSet Filter {
29       get {
30          if (cached)
31             return cachedFilter;
32
33          cached = true;
34          cachedFilter = (BitSet)Produced.BitSet.Clone();
35          foreach (BitSetProxy bitsetProxy in Required)
36             cachedFilter.Or(bitsetProxy.BitSet);
37
38          return cachedFilter;
39       }
40    }
41
42    /// <summary>
43    /// Determine if this rule is relevant to reach the given goal.
44    /// </summary>
45    public bool IsRelevantTo(BitSet goal) {
46       return Filter <= goal;
47    }
48
49    /// <summary>Guides the search for this rule instance.</summary>
50    /// <param name="plans">PlanSet to add plans to (also defines goal properties).</param>
51    /// <param name="limit">Abort the search if passing this cost limit (pruning).</param>
52    public abstract void Search(PlanSet planSet, ICost limit);
53
54    /// <summary>Id for this rule as given by the optimizer.</summary>
55    public int Id { get; set; }
56
57    public abstract bool StructurallyIdentical(IProducerRule other);
58
59    /// <summary>
60    /// The default implementaion is empty as many rules do not produce attributes.
61    /// </summary>
62    public virtual void MapShareEquivalentAttributes(IProducerRule from,
63                                                     Dictionary<NodeAttribute, NodeAttribute>
64                                                        equivalenceClassAttributeMappings)
65    { }
66    #endregion
67 }
```

Listing B.22: AbstractSearchRule implementation.

B.11 OrderManager's PrepareOrders

To illustrate how the ordering and grouping finite state machines are created, we have included the main method of the OrderManager below.

```
/// <summary>
/// Creates the finite automation and fills orderings and grouping proxies.
/// Call this after all interesting orders and groupings have been established
/// and before using any of them.
/// </summary>
public void PrepareOrders() {
   OrderingState dummy = NullOrderingState;

   // Find all mentioned properties
   determinePropertyDomain();
   // Filter unused FDs.
   filterDependencies();

   // Set ids of order and grouping proxies.
   int orderCount = 0;
   foreach (OrderProxy orderProxy in orderProxies.Values)
      if (orderProxy.Description.Size > 0)
         orderProxy.Id = ++orderCount;
   foreach (GroupingProxy groupingProxy in groupingProxies.Values)
      groupingProxy.Id = ++orderCount;

   // Construct the NFA
   NFA nfa = new NFA();
   NFANode nfaRoot = nfa.GetOrCreateNode(new OrderDescription());
   // Add a node with and edge from the root for each interesting order.
   foreach (OrderProxy orderProxy in orderProxies.Values) {
      if (orderProxy.Description.Size > 0) {
         NFANode current = nfa.GetOrCreateNode(orderProxy.Description);
         current.Id = orderProxy.Id;
         nfaRoot.AddEdge(new Edge<NFANode>(current, dependencyCount + current.Id,
                                  orderProxy.Description.ToString()));
      }
   }
   // Add a node with and edge from the root for each interesting grouping.
   foreach (GroupingProxy groupingProxy in groupingProxies.Values) {
      NFANode current = nfa.GetOrCreateNode(groupingProxy.Description);
      current.Id = groupingProxy.Id;
      nfaRoot.AddEdge(new Edge<NFANode>(current, dependencyCount + current.Id,
                                  groupingProxy.Description.ToString()));
   }

   // Graph nfa1
   GraphCurrentState(nfa);
   // Create equivalence classes for attributes.
   mapEquivalentProperties();
   // Create equivalence classes for orderings and groupings
   mapEquivalentOrderings();
   mapEquivalentGroupings();
   calculateGroupingInfo();

   // Add FD transitions for orderings
   nfa.ForAllNodes(considerTransitions, null);
   // Add FD transitions for groupings
   nfa.ForAllNodes(considerGroupingTransitions, null);
   // Graph nfa2
```

```
56      GraphCurrentState(nfa);
57
58      // Add share equivalent edges and graph nfa3.
59      if (ShareEquivalentMappings != null)
60         nfa.AddShareEquivalencyEdges(ShareEquivalentMappings);
61      GraphCurrentState(nfa);
62
63      // Add transitive and grouping epsilon edges to the NFA and graph nfa4.
64      nfa.AddEpsilonEdges();
65      GraphCurrentState(nfa);
66
67      // Optimize the NFA and graph nfa5.
68      nfaRoot.Id = orderCount + 1; // Just to make sure it is not removed by the next step
69      nfa.PruneArtificialNodes();
70      nfaRoot.Id = 0;
71      GraphCurrentState(nfa);
72
73      // Convert to a DFA and visualize
74      DFA dfa = new DFA();
75      DFANode dfaRoot = dfa.Convert(nfa, nfaRoot);
76      if (Visualizer.Visualize)
77         Visualizer.GenerateGraph(dfa.GetDot(), "dfa1");
78
79      // Fill proxies
80      dfaRoot.Order = new Order(0, new HashSet<OrderDescription> {new OrderDescription()});
81      NullOrder = dfaRoot.Order;
82      foreach (OrderProxy orderProxy in orderProxies.Values) {
83         if (orderProxy.Id == 0) {
84            orderProxy.Order = NullOrder;
85         }
86         else {
87            // Fill by using the edge from the root.
88            foreach (Edge<DFANode> edge in dfaRoot.Edges) {
89               if (edge.Label == (dependencyCount + orderProxy.Id)) {
90                  edge.Target.Order = new Order(orderProxy.Id,
91                                         new HashSet<OrderDescription> {orderProxy.Description});
92                  orderProxy.Order = edge.Target.Order;
93               }
94            }
95         }
96      }
97
98      // Fill grouping proxies using the edges from the root.
99      NullOrder = dfaRoot.Order;
100     foreach (GroupingProxy groupingProxy in groupingProxies.Values) {
101        foreach (Edge<DFANode> edge in dfaRoot.Edges) {
102           if (edge.Label == (dependencyCount + groupingProxy.Id)) {
103              edge.Target.Order = new Order(groupingProxy.Id,
104                                     new HashSet<NestableHashSet<string>> {groupingProxy.
                                          Description});
105              groupingProxy.Order = edge.Target.Order;
106           }
107        }
108     }
109
110     // Create orders
111     foreach (DFANode node in dfa)
112        if (node.Order == null)
113           node.Order = new Order(++orderCount,
```

```
114                         new HashSet<OrderDescription>(
115                            node.Description.Select(nfanode => nfanode.OrderDescription)));
116     // Create compatibility matrices
117     foreach (DFANode dfaNode in dfa) {
118        dfaNode.Order.Compatibility = new BitArray(orderCount + 1, false);
119        foreach (NFANode nfaNode in dfaNode.Description)
120           dfaNode.Order.Compatibility[nfaNode.Id] = true;
121        dfaNode.Order.Compatibility[dfaNode.Order.Id] = true;
122        dfaNode.Order.Compatibility[NullOrder.Id] = true;
123     }
124
125     // Compare artificial orderings for subsets to increase compatibility.
126     foreach (DFANode dfaNode1 in dfa)
127        foreach (DFANode dfaNode2 in dfa)
128           if (dfaNode2.IsSubsetOf(dfaNode1))
129              dfaNode1.Order.Compatibility[dfaNode2.Order.Id] = true;
130
131     // Create transition functions
132     foreach (DFANode node in dfa) {
133        node.Order.Transitions = new Order[dependencyCount];
134        foreach (Dependency dependency in dependencies.Values) {
135           foreach (Edge<DFANode> edge in node.Edges) {
136              if (dependency.Id == edge.Label) {
137                 node.Order.Transitions[dependency.Id] = edge.Target.Order;
138                 node.Order.PossibleTransitions |= (1 << dependency.Id - 1);
139              }
140           }
141        }
142     }
143     // Finally visualize.
144     if (Visualizer != null)
145        Visualizer.GenerateGraph(dfa.GetDot(), "dfa2");
146  }
```

Listing B.23: OrderManager.PrepareOrders.

C

Digital Appendix

"Any code of your own that you haven't looked at for six or more months might as well have been written by someone else."

— Eagleson's Law

The accompanying digital appendix includes this report, as well as the complete source code for our implementation. It is structured as follows:

/Report This report as PDF.

/Source Source code for the optimizer.

- **/bin** Dependent binaries, such as NUnit, Graphviz and so on.
- **/Optimizer** Source code for the optimizer.
- **/OptimizerProfiling** Console application used as an entry point to the optimizer for profiling.
- **/OptimizerTests** Unit and system tests of the optimizer.
- **/OptimizerTransformer** MARS query pipeline operator. This component is injected into the MARS query pipeline, invoking the optimizer when a query hits it.

/DebugOutput Debug output from the optimization of the largest example query in Chapter 7.

- **stats.txt** Timing statistics for the optimization.
- **plans.txt** All plans generated during plan generation.
- **rules.txt** Instantiated constructive rules.
- **search.txt** Search debug output from plan generation, showing how the search progresses.
- **shareequivalence.txt** Share equivalence classes.
- **before.dot/.pdf** Query before optimization.
- **preproc.dot/.pdf** Query after pre-processing, before plan generation.
- **planning.dot/.pdf** Query after plan generation, before post-processing.
- **after.dot/.pdf** Query after optimization, after post-processing.
- **nfa/dfa*.dot/.pdf** Graphs generated by the order manager during preparation, numbered stepwise.

We would recommend starting with `Optimizer/Engine/QueryOptimization.cs` if looking into the code.

Unfortunately, the source code as supplied will not build, as it depends on assemblies from *fast*, which we were not allowed to distribute. For the same reason, we are not able to provide running binaries of the optimizer.

To be able to test the optimizer, please contact any of the authors or Øystein Torbjørnsen at Microsoft (`Oystein.Torbjornsen@microsoft.com`).

Bibliography

[AT 08] AT and T Research. Graphviz - graph visualization software. http://www.graphviz.org/, 2008.

[BGPS01] Jun Rao Bruce, Lindsay Guy, Lohman Hamid Pirahesh, and David Simmen. Using eels, a practical approach to outerjoin and antijoin reordering. *Data Engineering, International Conference on*, 0:0585, 2001.

[Bil] Keith Billings. A tpc-d model for database query optimization in cascades. http://web.cecs.pdx.edu/~kgb/t/title.shtml.

[BN08] Alex Brasetvik and Hans Olav Norheim. iad: Query optimization in mars. http://folk.ntnu.no/brasetvi/master_preproj.pdf, 2008.

[Bra03] Kjell Bratbergsengen. *TDT4225: Lagring og behandling av store datamengder*. 2003.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. pages 150–160, 1994.

[CGK05] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Efficient algorithms for pattern matching on directed acyclic graphs. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 384–385, Washington, DC, USA, 2005. IEEE Computer Society.

[CMN98] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction: How much is enough. pages 436–447, 1998.

[Cor08] Microsoft Corporation. Sql server 2008 books online. http://msdn.microsoft.com/en-us/library/bb522495.aspx, 2008.

[CR94] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. *SIGMOD Rec.*, 23(2):161–172, 1994.

[CZ98a] Mitch Cherniack and Stan Zdonik. Changing the rules: Transformations for rule-based optimizers. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 1998.

[CZ98b] Mitch Cherniack and Stan Zdonik. Inferring function semantics to optimize queries. In *In Proc. of 24th VLDB Conference*, pages 239–250, 1998.

[GD87] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 160–172, New York, NY, USA, 1987. ACM.

[Gei08] Rubino Geiss. Grgen.net. `http://www.grgen.net`, 2008.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. *SIGMOD Rec.*, 21(2):9–18, June 1992.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[Gra95] Goetz Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[HFC+00] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[HKL+08] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, 2008.

[HR] Gerhard Hill and Andrew Ross. Reducing outer joins. *The VLDB Journal.*

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, 1993.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[ISA+04] IF Ilyas, R Shah, WG Aref, JS Vitter, and AK Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 203–214, 2004.

[LM94] Alon Y. Levy and Inderpal Singh Mumick. Query optimization by predicate movearound. In *In Proceedings of the 20th VLDB Conference*, pages 96–107, 1994.

[LP97] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[LPK+94] C. A. Galindo Legaria, J. Pellenkoft, M. L. Kersten, Arjan Pellenkoft, and Martin Kersten. Cost distributions of search spaces in query optimization, 1994.

[Moe06] Guido Moerkotte. *Building Query Compilers (Draft).* 2006.

[MS99] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[Neu05] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, Mannheim, 2005.

[NM04] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 960–971. VLDB Endowment, 2004.

[NM08] Thomas Neumann and Guido Moerkotte. Single Phase Construction of Optimal DAG-structured QEPs. `http://pi3.informatik.uni-mannheim.de/~moer/Publications/MPI-I-2008-5-002.pdf`, June 2008.

[NUn08] NUnit.org. Nunit - unit testing framework for .net. `http://www.nunit.org`, 2008.

[ONK+95] Fatma Ozcan, Sena Nural, Pinar Koksal, Mehmet Altinel, and Asuman Dogac. A region based query optimizer through cascades optimizer framework. *Bulletin of the Technical Committee on Data Engineering, Vol*, 18:30–40, 1995.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *In SIGMOD*, pages 39–48, 1992.

[Pos08a] PostgreSQL Global Development Group. Postgresql 8.3.4 documentation. `http://www.postgresql.org/docs/manuals/`, 2008.

[Pos08b] PostgreSQL Global Development Group. Postgresql 8.3.4 source code. `http://www.postgresql.org/ftp/source/v8.3.4/`, 2008.

[Pos08c] PostgreSQL Global Development Group. Postgresql history. `http://www.postgresql.org/about/history`, 2008.

[Red08] Red Gate Software Ltd. Ants profiler - .net code and memory profiler. `http://www.red-gate.com/products/ants_profiler/index.htm`, 2008.

[Roy98] Prasan Roy. Optimization of dag-structured query evaluation plans, 1998.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, It. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 23–34, 1979.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[SH05a] M Stonebraker and J Hellerstein. Anatomy of a database system. *Readings In Database Systems*, 2005.

[SH05b] M Stonebraker and J Hellerstein. What goes around comes around. *Readings In Database Systems*, Jan 2005.

[SRH86] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The design of postgres. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986.

[SSM96] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 625–628, London, UK, 1996. Springer-Verlag.

[Sto87] Michael Stonebraker. The design of the postgres storage system. pages 289–300, 1987.

[Wik09] Wikimedia Foundation. Wikipedia. `http://www.wikipedia.org/`, 2009.

[WLB03] Ju Wang, Jinmiao Li, and Greg Butler. Implementing the postgresql query optimizer within the opt++ framework. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, page 262, Washington, DC, USA, 2003. IEEE Computer Society.

[ZLFL07] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544, New York, NY, USA, 2007. ACM.