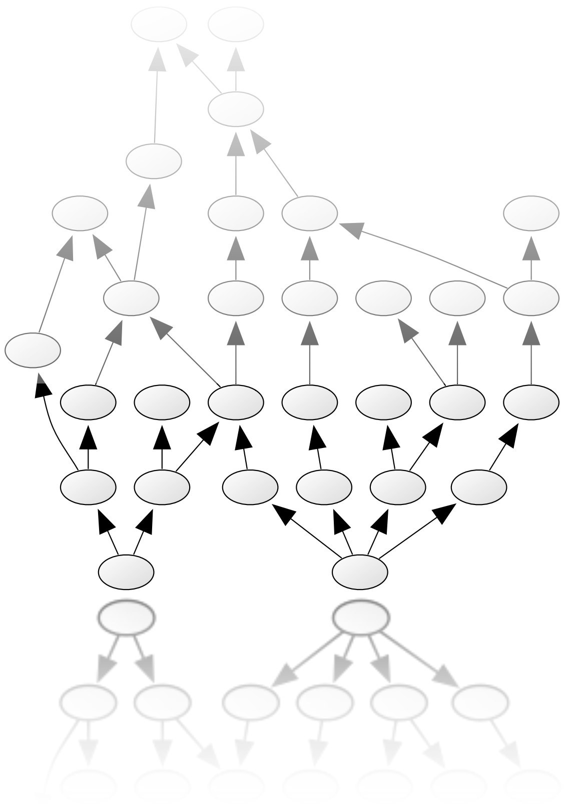
TDT4510 - Specialization Project

iAD: Query optimization in MARS

Trondheim, Fall 2008

Alex Brasetvik
Hans Olav Norheim

Supervisor: Svein Erik Bratsberg



NTNU
Norwegian University of
Science and Technology

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 the pre-project to the master thesis at the Department of Computer and Information Science at the Norwegian University of Science and Technology, autumn 2008.

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* which means that the optimizer must do so too. This turned out to be a greater task than what it might seem like. The optimizer also needed to be extensible, including the ability to deal with query operators it does not know, as well as supporting arbitrary cost models.

During the course of the project, we have laid out the design of an optimizer we believe satisfies these goals. DAGs are currently not *fully* supported, but the design can be extended to do so. 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.

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 go deep in this exciting topic.

During the course of the project we have learned a lot, and at the same time produced something we believe is a good foundation for the upcoming master thesis.

We would like to thank Svein Erik Bratsberg 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, and Guido Moerkotte at the University of Mannheim for providing us the latest draft of his query optimization book.

Alex Brasetvik

Hans Olav Norheim

Trondheim, December 2008

Contents

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 out of 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 1.2 and 1.3. In short, a query evaluation plan is a combination of operators that are actually executable by the evaluation system.

For example

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

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

are 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 TPC-* compared to rivaling products.

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

In this project, we have studied research articles related to query optimization, as well as real implementations, as implemented in Open Source DBMS-es such as PostgreSQL, MySQL, SQLite and MonetDB.

1.1 Goals of The Project

The goals of this project are to get a broad overview of the current state of query optimization, and sketch an extendable (as explained in Section 5.1) architecture that can serve as the basis for future development. We plan to continue the work done in this project in our master thesis the following semester.

Prioritized, our goals are 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 thesis, and not be so simple it needs to be replaced completely.

1.2 Abstract View of an Optimizer

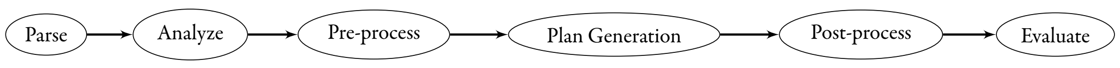


Figure 1.1: Overview of query processing

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, and **evaluation** for the step where the query plan is evaluated to produce result sets.

First, the query is parsed into a query graph. Normally, this query graph is a tree, but DAGs are also applicable query graph structures, as discussed in Chapter 4.

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

A valid query is then rewritten: Views and sub-selects are flattened, stored procedures inlined, etc. Then, the optimizer decides join ordering and -algorithms, pushes and pulls predicates around, and other techniques 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.

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 3

1.3 Runtime System

The output of the query optimizer is a **query evaluation plan** (hence "*query plan*" or QEP for brevity), which is executed by the query evaluator. This section briefly describes how typical **tree-structured query plans** are evaluated. In Section 4.3.2 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.

A query plan[2] is a tree structure where the nodes are **physical operators**. I.e. instead of containing relational algebra operators, such as one selection and a join, they contain two scan-operators and a specific join-implementation — for example a file scan, an index scan and a nested loop join.

The nodes typically have an **iterator-interface**. Evaluation is then done by consecutively calling `next()` from 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. Data typically originate from the leaf nodes, which are usually some kind of *scan*-node. However, this is not always the case, for example with MARS' *Exchange*-operator.

Some operators can be **pipelined**. That is, they pass on output incrementally, 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. When operators can be pipelined, the overhead of materialization is avoided. **Materialization** is saving the output (or input, if it is a separate node) 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 4.3.2, when output of one operator can be the input of multiple other operators, pipelining becomes more difficult.

1.4 What Makes MARS Different to RDBMS-es?

1.4.1 Introduction to MARS

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 `JOIN`s. *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. In MARS, data is structured into records, contained in *indexes*. These indexes can be joined, either via merge- or hash-joins — nested loop joins are currently not available. The indexes do not have any metadata about relations and foreign keys, and referential integrity and such is not enforced.

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.

1.4.2 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

[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.

output of one operator to be the input of more than one operator, introducing many optimization opportunities and -problems. See Chapter 4 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 usually done in *batches*, as updates are expensive and 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 Multi-query, which also means that the query optimizer must support it.

1.5 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 exists a HybridHashJoin operator and a MergeJoin operator, but no Join operator, which is the logical equivalent. Some of them are both logical and physical, e.g. Select or Project. Moreover, it currently does not store any statistics usable for query optimization.

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 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 project.

However, it is important to be aware of practical ways of getting better query plans, to design an extensible and maintainable query optimization framework that allows new rules and transformations to be developed later on — i.e. future-proofing the architecture.

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. 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) — 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 1.2. 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 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 1.3

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 has no restrictions on join inputs, and gives 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.

So far, our optimizer enumerates all bushy plans, and can do only left-deep plans, but not automatically if it realizes that there are too many relations.

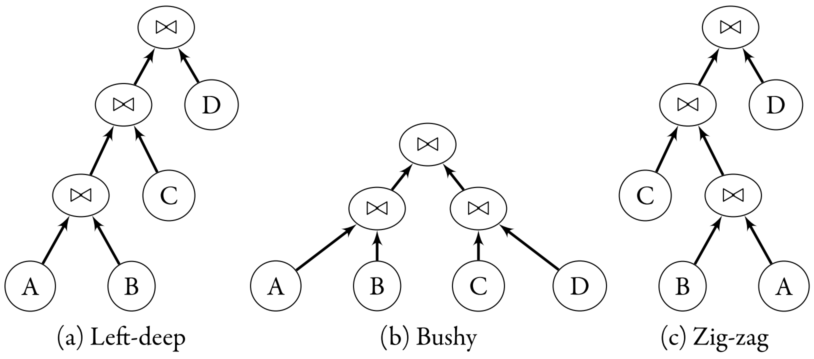


Figure 1.2: Three query trees

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[3] $\sigma_{\text{person.id}=42}\,(\text{person} \bowtie \text{city})$, 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[4]:

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

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.

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

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

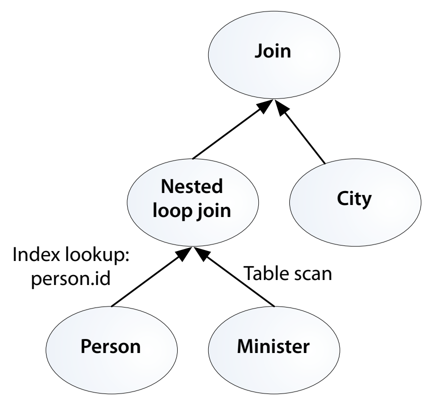


Figure 1.3: Example of a physical evaluation plan

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 1.3.

1.6.4 Statistics Maintenance and Cost Estimation

When deciding what access paths to use and in which way to order joins, the planners 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 3.

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. There are issues related to replication as well — which are also deferred to the next semester's master thesis.

1.6.6 Heterogenous 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 are their *extensibility*. Users can develop custom functions that are executed *on* the database.

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)`

The first two cases pose a challenge to an optimizer. The set returning functions *may* be optimized, if they are written in a procedural language native to the database and inlined during rewriting. Then they are optimized as a regular subquery.

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

```sql
/* 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 is 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.

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 and stops as soon as the top-k results can be reported. They 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 1.4.

Since MARS is a search engine that deals with information retrieval and not solely a database, the result sets are often ordered by some *rank*. MARS does not currently have rank-join-operators, so we have not delved deeply into them. 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 6.3.6.

1.6.9 Multi-query Optimization

We often need to perform multiple queries to get all the results 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 for every ordering we need them in, but few query optimizers considers 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 1.4. 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 4.2, 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.

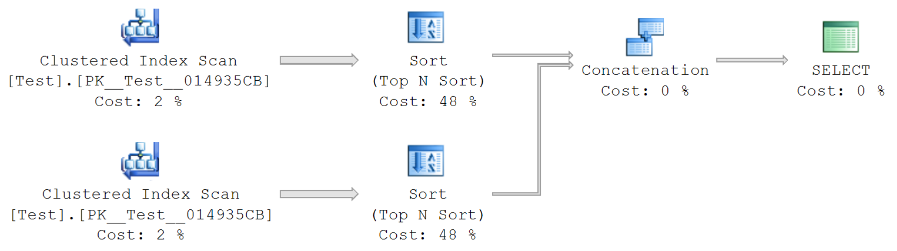


Figure 1.4: Example query tree which could be optimized in a query-DAG (screenshot from Microsoft SQL Server)

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[5], with the 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)

... 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.

[5]Example due to [CZ98b]

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

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 affects 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 3.

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 ≥ 12 [Pos08a].

A genetic optimization algorithm uses a nondeterministic, 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 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 gives and introduction to costing of query plans and ends with a description of the cost component in our optimizer. Chapter 4 discusses how MARS's DAGs affect query optimization. Chapter 5 is the bulk of the report, and describes our optimizer implementation, both design and algorithms. This flows naturally over to Chapter 6, which presents the rules the optimizer implementation uses. We discuss the rule interface, and have included a few samples. Chapter 7 presents the results of the project and discusses the current state of the optimizer. Chapter 8 concludes the report and wraps up further work and a plan for the next semester's work on our master thesis. Appendix A includes selected code samples, while Appendix B describes the contents of the accompanying CD-ROM.

Code Samples

The report includes quite a few code samples to illustrate how the optimizer implementation works. They are all 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, emphasizing 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 accompanying CD for the complete code.

In the code, we *do* follow C# coding guidelines (like curly braces on a new line), but has sacrificed it to save space in the report.

2 Case Studies and Previous Work

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, as it is 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*.

2.2 The Early Years: System R

As mentioned in the introduction, System R pioneered query optimization, proving that declarative, easy-to-use query languages were viable means of interfacing database systems. Its design choices has 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 subqueries are treated as subroutines which return tuples to the predicates they occur in. Queries are not rewritten to *flatten* subqueries, 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. This definition of "interesting" is a bit limiting, though. It is possible to exploit orderings also when the query itself does not imply a specific ordering — for example when two clustered indexes are available for two relations that are to be joined, possibly making a merge join a cheap alternative. The cheapest plans are kept for further consideration.

To be able to estimate approximate costs for access paths, *statistics* about the various relations are fetched from the system catalog. The statistics are 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 subquery`. We omit them for brevity.

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	NINDX (i) + TCARD
Non-clustered index i not matching any predicate	NINDX (i) + NCARD
Same, but small enough to fit in memory	NINDX (i) + TCARD

Table 2.2: Estimated number of page fetches in System R

[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.

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 bind 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. The project was started as a research project by Michael Stonebraker, as "Postgres" at the time — a followup project 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 subquery flattening as the most interesting feature. We did not prioritize achieving breadth in the study of *implementations*. The optimizers in the mentioned projects 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 6.2.3. The study was something we started out with in the initial phase of the 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 would not directly apply when having to deal with DAG-structured query plans*. 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. 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 could 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 6.2.3. They constitute just a fraction of the total optimization time for queries with many relations.

The rules of the **search phase** determine how the search space is explored. Thus, these obviously need to be treated efficiently, and 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 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 *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* $\rightarrow$ {*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, and providing more efficiency 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 prune bad paths early. As EXODUS, it used a model to generate code, which in turn was linked to the implementer's database system, as well as having 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 subtree
- Incremental enumeration of the search space
- Optimization tasks as data structures

A rule object can be created and optionally modified (disabled, reconfigured, etc.) at runtime. 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 5.6. 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 subtree, with a subsequent phase that made physical plans from these and compared them to each other. The exploration is governed by heuristics that avoids repeatedly exploring the same subspace. 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, which in turn can result in pruning the rest of the 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 based on the Volcano-optimizer mentioned in Section 2.5.1. It uses two steps. First, the operators that might 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 in [Neu05]:

- Identifying sharable 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 equijoins!

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

We are unaware of any other *transformative* approach to optimizing DAG-structured query evaluation plans.

2.5.3 Constructive: Starburst and Neumann/Moerkotte

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

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 produces 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, as well as a 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 5, 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 7, 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 subplans 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 subgoals 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 subproblems. 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 subgoals), only subplans satisfying subgoals 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 subgoals 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 subproblems, if this problem is not mitigated. It is easily remedied by checking if the subproblem 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 7.4 on page 68, 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 5.5, the problem is ameliorated by caching the answers to `GoalIsUnreachable(goal)`. By disabling the cache, a large query which took 4.72 seconds to optimize with caching took 13.42 seconds.

Another advantage with the bottom-up approach is avoiding a lot of cache lookups for the same subproblems 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. In this stage in the project, time is spent better 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 project 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 4.3.2.

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 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 5 and the rest of the report is devoted to our changes to the model and the design- and implementation details.

3 Cost Estimation and Statistics

3.1 Introduction

The goal of query optimization is to find the *best* plan (or one that is reasonable close to it), which is the *cheapest* one 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 result sets — 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.

We have not had time to study cost- and statistics issues thoroughly so far. However, we have looked into some issues, 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 right 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.

In Section 3.2 we describe what influences the cost, and partially why it is important to abstract the calculations. Section 3.3 shows some example uses of statistics and how it influences the planning process. Then, Section 3.4 describes why a cost *component* makes sense, and what we want from it. Lastly, Section 3.6 wraps up about the current status of MARS with respect to costs and statistics.

3.2 Cost Factors

To calculate a plan's cost, several factors are considered. Available buffer space, amount of IO 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 what it needs to do its task. It can estimate the buffer sizes and the amount of memory it needs, expected random and sequential needs 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". For example, if a conventional disk is replaced with an SSD-disk that provides cheaper random reads, the cost is reduced. 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 cost of an operation:

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

- **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.

The next section discusses how an operator can estimate parts of its costs.

3.3 Statistics and Example Calculations

To be able to guesstimate costs, statistics about the various relations are used. In Section 2.2, we described how System R estimates cardinalities and selectivities — and its limitations. Since then, a lot of effort has 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].

First, we show some examples on how statistics are used to estimate selectivities. In Section 3.5 we go through some issues related to how the statistics are gathered and maintained.

3.3.1 Example Calculation of Selectivity Estimation

The following examples are adapted from PostgreSQL's documentation about how their planner uses statistics [Pos08a, Ch. 55]. They demonstrate how histograms and most common value lists can be used — in isolation and combined. We assume familiarity with histograms and their various representations.

The example assumes a table `tenk1`, which is part of the regression test database in PostgreSQL 8.3. `tenk1` has 10 000 tuples (`reltuples`) contained in 358 pages (`relpages`). These statistics are stored in the system catalog, and updated occasionally during a `VACUUM` or `ANALYZE` — two maintenance commands. They are used to estimate the real number of tuples in the table (a count which is expensive to maintain accurately), which is interpolated if the actual page count (a count that is cheap to maintain) differs from the page count that corresponds to the last tuple count. When performing the query `SELECT * FROM tenk1 WHERE unique1 < 1000`, the planner looks up the selectivity function for the `<`-operator, and the histogram for the column `unique1` in `tenk1`, which is

$$\text{buckets} = \{0, 993, 1997, 3050, 4040, 5036, 5957, 7057, 8029, 9016, 9995\}$$

These values are used to determine the selectivity of the predicate `> 1000`. The histogram is an equidepth histogram, so selectivity is determined by locating the bucket the value is contained in, and then count part of it and all the proceeding buckets:

$$\begin{aligned}\mathcal{S} &= \frac{1}{|\text{buckets}|}\left(1+\frac{1000-\min(\text{buckets}_2)}{\max(\text{buckets}_2)-\min(\text{buckets}_2)}\right)\\ &= \frac{1}{10}\left(1+\frac{1000-993}{1997-993}\right)\\ &= 0.100697\end{aligned}$$

Note that two adjacent values in the buckets-list define a bucket, so $|buckets| = 10$ even though the length of the list is 11. With the selectivity, we can estimate the number of rows:

$$\begin{aligned}\text{rows} &= \text{cardinality} \times \text{selectivity}\\ &= 10000 \times 0.100697\\ &= 1007\end{aligned}$$

This was with a range predicate. What about an *equality* predicate? A list of common values is used. Assume the list of the 10 most common values (MCV) and their respective frequencies for `tenk1.stringu1` are

$$\begin{aligned}&\text{mcv} = \{\\ &\qquad (\text{EJAAAA}, 0.00333333), (\text{BBAAAA}, 0.003), (\text{CRAAAA}, 0.003),\\ &\qquad (\text{FEAAAA}, 0.003), (\text{GSAAAA}, 0.003), (\text{JOAAAA}, 0.003), (\text{MCAAAA}, 0.003),\\ &\qquad (\text{NAAAAA}, 0.003), (\text{WGAAA}, 0.003), (\text{FCAAAA}, 0.003)\\ &\quad \}\end{aligned}$$

Also, we assume the fraction of values being `NULL` is 0 and the number of distinct values is 676. With the query `SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA'`, we find the value `'CRAAAA'` in the list, so the selectivity is simply 0.003, and the number of estimated rows is 30. But what with selectivity estimates for values *not* in the list? Then we estimate the selectivity as

$$\mathcal{S} = \frac{1-\sum \text{mcv}_{\text{freq}}}{\text{distinct}\,|\text{mcv}|}$$

$$= \frac{1-(0.00333333+9\times 0.003)}{676-10}$$
$$= 0.0014559$$

Thus, we can estimate that `SELECT * FROM tenk1 WHERE stringu1 = 'foo'` will return $10000 \times 0.0014559 = 15$ rows.

For *non-unique* columns, there will usually be *both* a list of most common values and a histogram — and the histogram disregards any values listed in the most common values. Selectivity estimation for a range predicate then uses both the values found in the MCV and the selectivity determined by the histogram damped by the fraction of histogram values not in the MCV:

$$\mathcal{S} = \sum\left(\mathcal{S}_{\text{relevant common values}}\right) + \mathcal{S}_{\text{histogram}} \times \text{histogram fraction}$$

In cases where some values are significantly more common than others, the histogram fraction damping factor will decrease correspondingly.

The query `SELECT * FROM tenk1 WHERE stringu1 = 'foo' AND unique1 < 1000` contains two predicates, whose selectivities we have estimated to $\mathcal{S}_{\text{stringu1}} = 0.0014559$ and $\mathcal{S}_{\text{unique1}} = 0.100697$. The selectivities are assumed to be independent, so the selectivity of `stringu1 = 'foo' AND unique1 < 1000` is thus $\mathcal{S}_{\text{AND}} = \mathcal{S}_{\text{stringu1}} \times \mathcal{S}_{\text{unique1}} = 0.0001466$. Without correlated histograms, that is the best one can do.

These examples also illustrate 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 is not severely offset by the costs.

Section 5.5.4 describes where in the process such computations would occur.

3.4 Cost Component

We have not yet had time to deeply study how a cost component should work, but we have given some thought into how we want to interface it.

3.4.1 Plan Generator Interface

First and foremost, the optimizer and the rules used do not care *why* a plan is better 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 3.1 shows the interface the optimizer expects a plan's cost to implement. At this point, a real cost component has not been developed, and just a simple one described in Section 5.5.3 is currently used. It uses a single number to model the cost. However, the interface remains the same.

Listing 3.1: Planner's cost model interface.

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

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

`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 one uses less memory and more I/O, it returns unknown. The decision is then left to the planner of whether or not to keep the plan. Often, both (sub)plans will be kept and the final plan decided upon when the optimization is done. 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.

3.4.2 Rule Interface

Above, we presented the cost component interface for the plan generator. However, the rules also need some kind of interface to the cost component to be able to model the cost 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 how to compare plans, the rules must be coded to utilize the cost model and the cost model must support the modeling needs of the rules. However, it still makes sense to create an interface between the rules and cost model that is as clean as possible.

We propose a cost model that takes all the cost factors previously mentioned into consideration, by allowing all rules to model its operator costs specifying the expected cost in each dimension. Even if not all of them are used to decide which plan is best in the end, they are available and may be used later. The cost of maintaining them is minimal — we propose to model each cost factor as a number.

The cost class could have a constructor with the following signature:

```
public Cost(int cpu, int sequentialReads, int randomReads, int memory, int network)
```

By overloading the + operator for the Cost class, a rule can then easily specify the costs of a plan. For example, a selection rule, requiring no I/O could then specify its cost as such:

```
Costs = inputCosts + new Cost( Cardinality * PredicateCost, 0, 0, 100, 0)
```

3.5 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 random sampling [CMN98]. 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. This is something we would like to look more into during the master thesis the next semester.

3.6 Statistics and MARS

MARS does not currently store any statistics, and the assemblies we were provided with only contained a simple FileReaderScan. 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 project — 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.

Thus, deeper studies of statistics have been deferred to the master thesis the next semester. However, by looking into how some systems use and gather statistics (as described previously in Section 3.3 and Section 3.5), we have an idea of what to look into.

4 DAG-Structured Query Graphs

4.1 Introduction and Previous Work

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.

Since MARS supports evaluation of DAG-structured query evaluation plans (DQEPs), and not that much work on optimizing them already exist, we wanted to make sure we laid out the design of something that can be extended for DAG support later on. As mentioned in Section 2.6, we have based most of our work on a PhD-thesis by Dr. Thomas Neumann. Even though what we have designed so far does not have full DAG support, basing the design on Neumann's DAG optimizer makes sure it can be extended to support it. This chapter contains explanations of advantages and difficulties with DAG-queries, with many references to his work. It serves as a foundation to the topics dealt with in the subsequent chapters — design, implementation and rules.

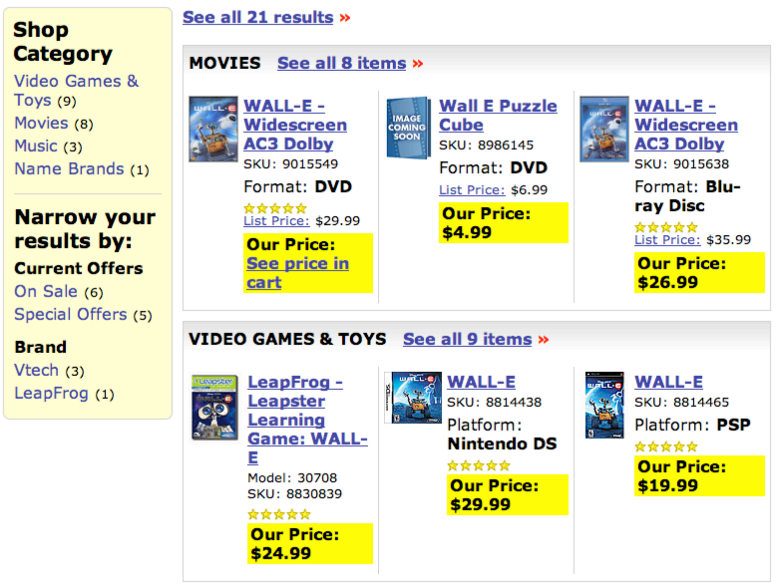
We start with a motivating example in Section 4.2. Section 4.3 describes some of the challenges with DQEPs, while Section 4.4 discusses *share equivalence*.

4.2 Motivation

Using DQEPs introduces many challenges compared to those that are tree-structured, so a motivating example to see that 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.

Consider for example the query shown in Figure 4.1. It is a search for "wall-e" on BestBuy.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. 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.



Shop Category
Video Games & Toys (9)
Movies (8)
Music (3)
Name Brands (1)

Narrow your results by:
Current Offers
On Sale (6)
Special Offers (5)

Brand
Vtech (3)
LeapFrog (1)

See all 21 results »

MOVIES See all 8 items »

WALL-E - Widescreen AC3 Dolby
SKU: 9015549
Format: **DVD**
List Price: $29.99
Our Price: See price in cart

Wall E Puzzle Cube
SKU: 8986145
Format: **DVD**
List Price: $6.99
Our Price: $4.99

WALL-E - Widescreen AC3 Dolby
SKU: 9015638
Format: **Blu-ray Disc**
List Price: $35.99
Our Price: $26.99

VIDEO GAMES & TOYS See all 9 items »

LeapFrog - Leapster Learning Game: WALL-E
Model: 30708
SKU: 8830839
Our Price: $24.99

WALL-E
SKU: 8814438
Platform: **Nintendo DS**
Our Price: $29.99

WALL-E
SKU: 8814465
Platform: **PSP**
Our Price: $19.99

Figure 4.1: Example of search that results in multiple smaller searches

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 4.2, 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 4.1.

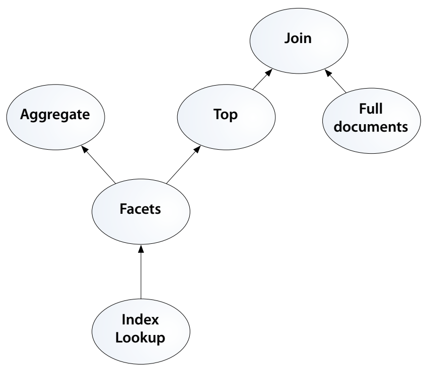
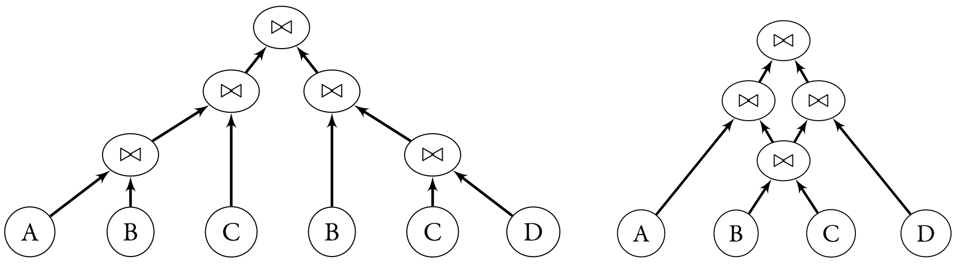


Figure 4.2: Imaginary DAG-structured Query Evaluation Plan



(a) Join tree (b) Join DAG

Figure 4.3: Two equivalent join graphs

4.3 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.

4.3.1 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 [1], consider the query $\mathsf{A} \bowtie \mathsf{B} \bowtie \mathsf{C} \bowtie \mathsf{B} \bowtie \mathsf{C} \bowtie \mathsf{D}$. Figure 4.3a shows a possible solution where the optimizer first finds that $(\mathsf{A} \bowtie \mathsf{B}) \bowtie \mathsf{C}$ and $\mathsf{B} \bowtie (\mathsf{C} \bowtie \mathsf{D})$ are partial optimal solutions, which are then combined. If the sub-optimal partial solutions $\mathsf{A} \bowtie (\mathsf{B} \bowtie \mathsf{C})$ and $(\mathsf{B} \bowtie \mathsf{C}) \bowtie \mathsf{D}$ had been considered, the *common sub-expression* $(\mathsf{B} \bowtie \mathsf{C})$ could have been shared, resulting in the plan in Figure 4.3b. Consequently, an optimal DAG cannot be constructed by simply combining optimal partial plans.

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 $(\mathsf{B} \bowtie \mathsf{C})$ 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. [Neu05] describes several algorithms that deal with this problem, but they are too complex to explain here, as we have not implemented DAG-costing ourselves.

4.3.2 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.

[1]Example adapted from [Neu05]

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 4.1 is shown in Figure 4.4. 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 is outside the scope of this project.

Listing 4.1: Sample query resulting in a shared temporary relation

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

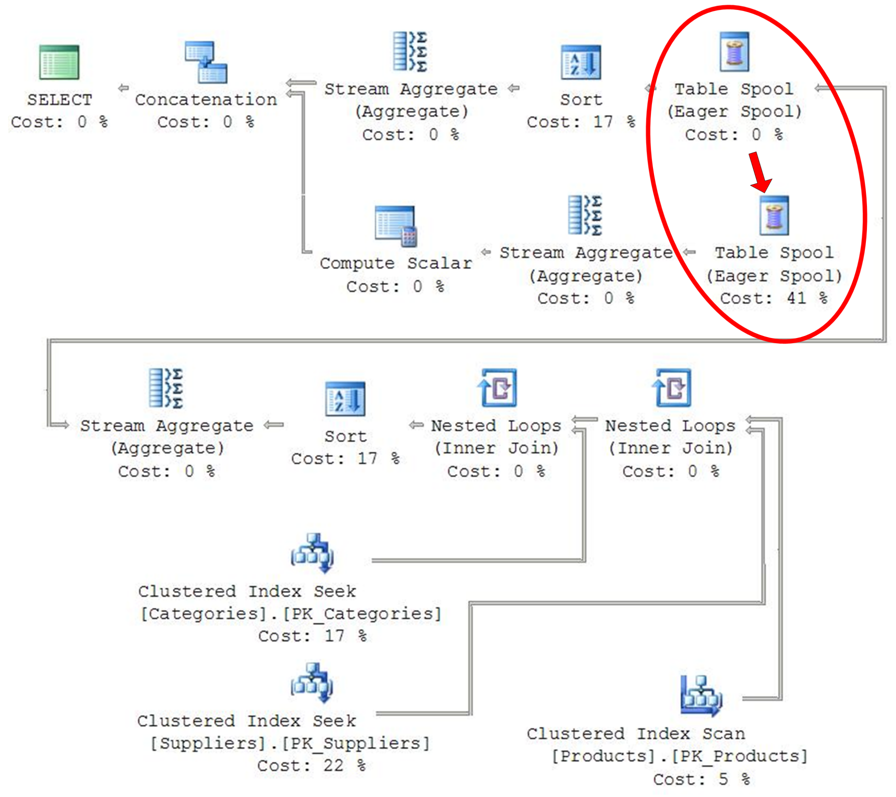


Figure 4.4: Example Plan generated by SQL Server 2008 sharing a temporary relation

4.4 Share Equivalence and Common Subexpressions

Two expressions are **share equivalent** *if one expression can be computed by using the other expression and renaming the result* [NM08]. It is used to detect if the same operations are performed twice in a plan — and should have its cost counted only once. The property is also used by the plan generator when checking if a subproblem has occurred before. It is unusual to see *exactly* the same subproblem twice in a query, but share equivalence is sufficient to share the partial results. However, we have not currently implemented the share equivalence check, as explained in Section 5.5.5 — therefore, we *currently* only get DAGs as output when *exactly* the same subproblems occur.

In [NHM05], it is shown that introducing **common subexpression elimination** makes the (comparatively) simple problem of ordering selections and maps NP-hard. In [NM08], it has been argued that this suggests that in a DAG-context, the decision about reusing intermediate results must be made by the plan generator. However, identifying and *using* identified common subexpressions are separate issues [RSSB00].

Exploiting share equivalence and common subexpressions are on the list of subjects to look more deeply into the next semester.

5 Design and Implementation

5.1 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. First, we go over our design goals for the query optimizer:

1. Extensibility
   (a) Support for arbitrary operators
   (b) Support for arbitrary cost models for operators
   (c) Support for arbitrary pre- and post-processing.
2. Clean design and implementation, exploiting what object-oriented programming gives
3. At least future support for DAG-structured query plans
4. Efficient plan generation

Extensibility was the most important goal with the project. The optimizer should have support for arbitrary operators, meaning that it should be able to add optimization rules for operators added after the optimizer was originally designed. To achieve this, the optimizer cannot know anything specific about the operators, but uses rules created by the operator implementer instead.

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 as this can be useful in specific uses of the optimizer.

We saw it as more important to come up with a good architecture and a clean design, than to implement as much functionality as possible. Therefore, we have only implemented rules for file scans, join and selection and put more focus on designing the optimizer core.

The rationale for supporting DAG-structured query plans was given in Section 4.1.

Efficiency has not been prioritized. Clearly, design decisions that prevents efficient execution must be avoided, but we have not delved into optimizing the implementation — for example, we have implemented a naïve pattern matcher, but with its declarative interface, it can be replaced with an efficient state machine generator later on.

5.1.1 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 eases 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 something bad did not happen (e.g. Exception) or that the resulting query is the optimal one. The tests have also been used to generate the results found in Section 7.

As an example, we have included the code for one of the plan generator tests in Section A.6. Test coverage has not been highly prioritized so far, though.

5.2 The Big Picture

As mentioned in Section 1.2, 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 invalidation. Such components depend heavily on the run time system, and is subject to further work when we start integrating our optimizer with MARS. As such, Figure 5.1 shows the parts of the query optimizer that actually optimizes the query, where our focus has been.

5.2.1 Pre-/post-processing vs Plan Generation

The unoptimized query enters the optimizer in the upper edge of the figure, 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. *The distinction between pre-/post-processing and plan generation is a very important one.* While pre-/post-processing is transformative and linear, plan generation is constructive and combinatorial. 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, 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 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

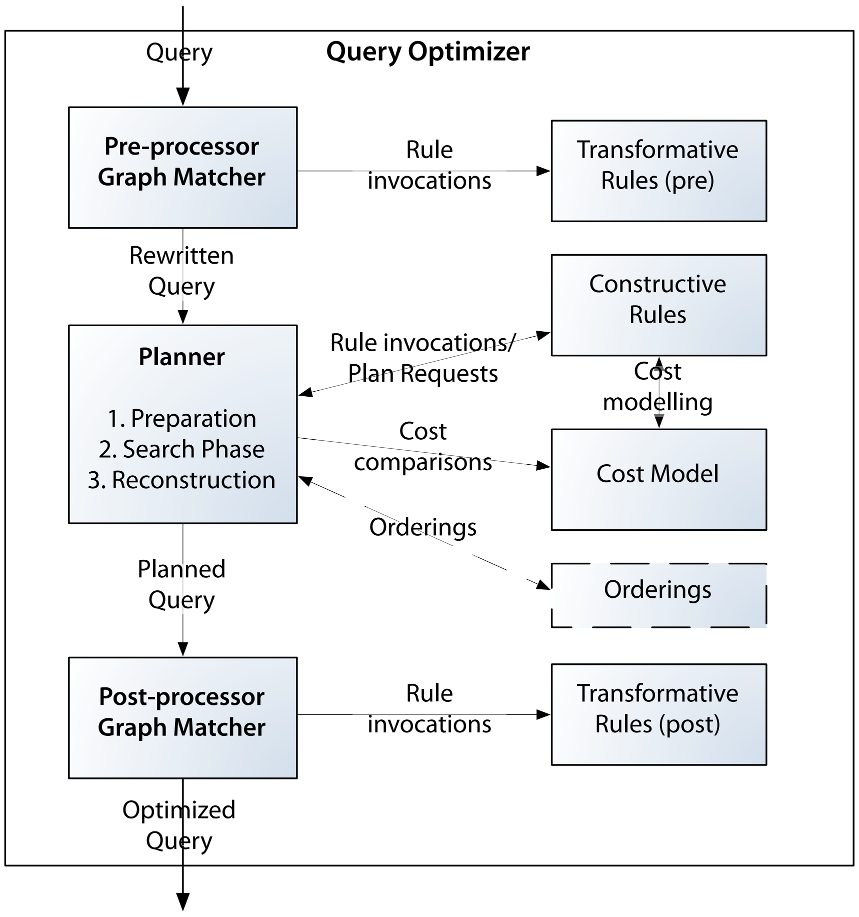


Figure 5.1: Query optimizer overview.

increase 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*.

5.2.2 Optimization Steps

In the **pre-processing** step, transformations like view flattening or predicate push-down or pull-up [LM94] is carried out. Rewrites are performed by applying transformation rules where the before-pattern match a subgraph of the operator graph.

After pre-processing, the operator graph is passed on to the **plan generation** step. Plan generation is the step that we traditionally perceive as query optimization. It substitutes physical operators for logical ones (e.g. HashJoin for Join), 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.

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. The goal is expressed as a *query goal property set*. Examples of properties in this set include "attribute X available" and "operator X 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 5.5.1.

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 controls the direction of the search. Basically, solutions to subproblems are combined into solutions of larger problems and finally the whole query, but in a top-down fashion. *Memoization*[1] 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 it back into the node structure used by the pre-/post-processing steps. This is carried out recursively by the rules.

As Figure 5.1 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. To keep track of different plans' useful orders (i.e. sorting), an **ordering component** will be used, but this is not implemented or designed yet, but we have a plan for it, see Section 5.5.2.

Finally, the planned query is **post-processed**. This is similar to pre-processing, but on physical algebra and typically, other types of rewrites are performed. Examples include merging of successive selection and map operators.

5.3 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 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 by 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 not be very extensible nor maintainable. Instead, we implemented our own *Node* class to model the graph, shown in Listing 5.1. Using a separate class also makes it easier to test the optimizer separately from MARS.

The property *OperatorType* stores the type of the MARS operator this node originated from, while the *Rules* list holds references to all rules that was instantiated for this node during the plan generation preparation phase. *Parents* and *Children* stores lists of the parents and children nodes of this node. Finally, the dictionary *Properties* allows us to store arbitrary properties in the node, identified by name. We also implement a C# indexer (basically we overload the [] operator) so we can access the properties like so: `someNode["Property"]`.

Listing 5.1: Node class (simplified, some code omitted)

```
public class Node {
    public Type OperatorType { get; set; }
    public List<IProducerRule> Rules { get; set; }
    public List<Node> Parents { get; private set; }
    public List<Node> Children { get; private set; }
    public Dictionary<string, object> Properties { get; set; }

    public object this[string name] {
        get { return Properties[name]; }
        set { Properties[name] = value; }
    }
}
```

[1]This is not a misspelling, the name *memoization* comes from *memo*.

As it looks now, when a query is received from MARS, it will be handed to the optimizer as an *IOperator* graph. The optimizer will have to traverse this graph from the root and translate it into a graph of *Nodes*. *OperatorType* will be set, parent and children relationships retained and all readable properterties will be copied into the *Properties* dictionary. When the optimization is complete, the *Node* graph will be translated back to an *IOperator* graph.

At this stage, we have not integrated the optimizer with MARS, so this process is subject to change.

5.4 Pre- and Post-Processing

Pre- and post-processing is mainly about query rewriting, but may also perform other tasks, like tagging the operator graph with information to be used later, for example to speed up the plan generation step. In the **pre-processing** step, transformations like view flattening, subquery merging/flattening, predicate push-down or pull-up [LM94] or different join transformation (like ANY/EXISTS $\rightarrow$ JOIN) is carried out. Other transformations that are "always smart to perform" are also carried out, like our Trim-Sort merge which we present in Section 6.2.2. Transformations removing unnecessary nodes can also be useful, as it reduces the running time of plan generation. Pre-processing operations that tag the graph nodes for later use in plan generation can also be employed.

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. 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. Each alternative will probably overlap fairly much, which means that much double plan generation 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. The data structures and algorithms in the current design do not support this, but we will look into extending the `Node` class to be able to express it. The preparation phase of plan generation will also need to be slightly extended.

Post-processing is similar to pre-processing, but on physical algebra and typically, other types of rewrites are performed. Examples include merging of successive selection and map operators, group-by push-down and our Trim-Sort merge may be applied again. Optimizations that does 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.

Only small parts of the pre- and post-processing steps (one rule and limited graph matching) have been implemented so far, so time will show if we need to alter the design presented here.

The processing is driven by a collection of rules that declares a *Pattern* it is looking for in the operator graph. For transformation rules, one may look at this pattern as the left-hand side (LHS) of a production, where the right-hand side (RHS) is the output when the rule is invoked. Transformation rules usually transform the query from one form to another, equivalent and "better" form. The rule does not need to transform the query graph — it may only add some information to it.

When the query optimizer finds that the pattern declared by a rule matches a part of the operator tree, it will invoke the rule with the context of the match (`PatternMatch`). We could have chosen to express the rules as productions, but instead chose to make all rules expose a `Fire` method which the optimizer calls when the rule is invoked, close to what was done in Startburst [PHH92]. This allows us to write more expressive rules that can reason about the match found and take the appropriate action. 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 below. For an explanation of how to express patterns and what `PatternMatch` is, see Section 5.6 on graph matching.

Listing 5.2: ITransformationRule interface

```
interface ITransformationRule {
    AbstractNodeMatcher Pattern { get; }
    void Fire(PatternMatch match);
    List<string> DependsOn { get; }
    bool Iterative { get; }
}
```

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 startup, all rules are topologically sorted by the optimizer and serves as a foundation for rule invocations.

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. If the rule returns *true* for `Iterative`, the optimizer will run this single rule until the graph stabilizes.

The plan is to have the optimizer apply rules successively until the graph converges to a final stable result. This makes it easy to implement rules, but may not yield the best running time, and we must be careful not to get in a "ping-pong" situation where the graph is transformed back and forth. Therefore, we plan to invest some time in studying [PHH92], which divides the rules into multiple classes that are applied as units. Rules can also choose to invoke other rule classes as part of their execution. This will probably yield better performance, but still, it is probably the plan generation step that will take the bulk of the time anyway.

5.5 Plan Generation

The design of the plan generation step is based on the work in [Neu05] and [NM08], and 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. We summarize the most important ones here, and fill in more details in the appropriate sections.

Left-deep join enumeration. The implementation of joins in the works above 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 6.3.6

Caching of unreachable plans. Often, the optimizer is asked to produce a plan that is not possible to create. In the works above, 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 13.42s to 4.72s**. See explanation of Listing 5.10.

Transformation rules. We have added transformation rules to enable pre-/post-processing and query rewriting. See Section 5.4.

Nicer implementation of BitSet (property set). In the works above, property sets are only viewed as bit sets. We have implemented a property set that behaves like a set of string (nicer to work with and debug), but still performs well and offer compact bit mask storage. This is described in Section 5.5.1.

Visualization of query plans. We have implemented query plan visualization. The result can be seen in Section 7.3.

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

Dynamic rule discovery. We have added the rule binder mechanism for dynamic rule loading by the optimizer, as described in Section 6.1.

Refactored interfaces. Some of the rule interfaces have been refactored (e.g. *IProducerRule*) to allow for cleaner implementation. See description under Section 6.2.1.

We also have more improvements we have not had time to implement yet, see Section 7.2 and Section 8.2.

In the plan generation phase, each rule constructs one part of the query. Each rule usually represents one logical operator, but 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 subproblem of the whole query, and finally the complete query. Each plan can have a number of *subplans*, solving a subproblem of the plan's problem (actually, its input).

The code below shows the main method of the plan generator, somewhat simplified. It closely resembles what was described in Section 5.2. First, a new *BitSetManager* is created to handle the property sets for this query. Then rules relevant to this query are initialized, goals determined (if this is a multi-query), 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 and returned.

Listing 5.3: Plan Generator main method, simplified

```
public Node Optimize(Node query) {
    // Create a new BitSetManager for this query
    BitSetManager = new BitSetManager();
    // Find and instantiate rules.
    InitializeRules(query);
    // Determined logical goals of query
    DetermineGoals(query);
    // Instantiate the memoization table
    plansCache = new Dictionary<BitSet, PlanSet>();
    // Make plans for leaf-nodes, i.e. scans.
    InitializeBasePlans();
    // Go plan!
    GeneratePlans();
    // Convert plans back to nodes.
    Node finalPlan = MakePhysicalPlan();
    return finalPlan;
}
```

To be able to illustrate how the plan generation works, we now introduce a simple query which we will use throughout this section. It is a simple join of two relations with a selection on an attribute of the second relation, expressed in SQL below. Figure 5.2 shows how this query might be passed to optimizer for optimization. The topmost operator is MARS' query operator.

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

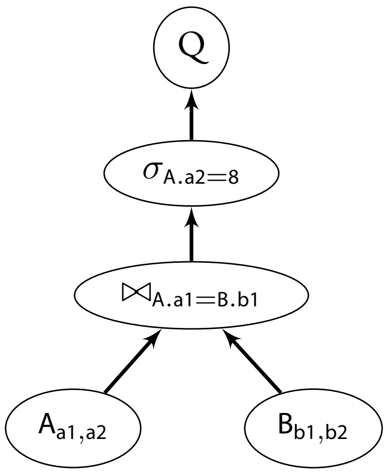


Figure 5.2: Sample query.

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.

5.5.1 Property Sets

During the plan generation phase, the plan generator generates many plans as solutions to sub-problems. In some way, it needs a way to annotate these plans with what they actually produce — 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 one *Required* set for each input (1 for rules representing unary operators, 2 for binary). 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 5.2 are given in Table 5.1. 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 $\{\bowtie\}$, that itself was applied.

The *global query goal* is the property set the final, complete query plan should satisfy and 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, all operators are logical, so the *global query goal* = $\{a1,a2,b1,b2,\bowtie,\sigma\}$.

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

Table 5.1: Instantiated rules for query in Figure 5.2 and produced/required sets.

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 ”`ATTRIBUTE_X`” and ”`APPLIED_Y`”.

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 plan, 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 are determined before the search phase begins. As [Neu05], we therefore assign a bit to each possible property and simply store the property sets as bit masks. This allows us to store 8 properties in just 1 byte. See Section A.5.1 for details. Further, this allows for *very* fast set operations, as set intersection is simply a bitwise AND between two property sets, probably performed in a single CPU instruction. See Listings A.12 and A.13 in Section A.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 5.4 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.

Listing 5.4: BitSet struct (simplified, interface only)

```
public struct BitSet : ICloneable {
    void Add(string property);
    bool Contains(string property);
    void Remove(string property);
    bool Overlaps(BitSet other);
     static bool operator ==(BitSet a, BitSet b);
     static bool operator <=(BitSet a, BitSet b);
     static BitSet operator |(BitSet a, BitSet b);
     static BitSet operator &(BitSet a, BitSet b);
     static BitSet operator -(BitSet a, BitSet b);
    void Or(BitSet other);
    IEnumerable<BitSet> Walk();
}
```

To conserve space, each BitSet does not store the mapping *string* $\longrightarrow$ *bit index*. That is responsibility of the *BitSetManager* class. How the Add, Contains and Remove method use it is shown in Listing A.10 in Section A.5.1.

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

BitSet Minimalization

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

5.5.2 Orderings, Groupings and Other Interesting Properties

When dealing with sub-plans, there are other properties that need to be considered than their raw cost. Costs are just half the story — properties such as ordering, groupings, subexpression sharing (for DAGs) and ranking 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" plan. One 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, it makes sense to not do it, since we need more reasoning capabilities for it and can represent it more compactly in a specialized form. The most common factor is *useful orderings*, as introduced by Selinger et al in [SAC$^+$79]. We continue by introducing orderings.

Orderings

There is more to orderings than one might think. A tuple stream has a single *physical ordering*, which is the actual order of the tuples. It can however have multiple *logical orderings*. For example, if a stream has the logical ordering (a, b, c), it also has (a, b) and (a). If the filter $a = c$ is applied to the tuple stream, this introduces the orderings (c, b, a), (c, b), (c, a), (c) and so on. If the filter $d = 4$ is introduced, we get the orderings (a, b, c, d), (d, a, b, c) and so on.

So far, we have not had the time to implement functionality to make the plan generator consider useful orderings and groupings. As such, the data structures and rules presented does not include this functionality, but we have put "todos" where it will be inserted. Most prominent is the *Order* property in the *Plan* class. This property contains an object describing the orders the plan produces. 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.

The *Order* property will contain an order state object for the plan. Additionally, the order component shown in Figure 5.1 is needed to be able to reason about the orderings. [Neu05] implements this as a finite state machine. It supports the two operations *ContainsOrder(logicalOrder)* and *Infer(...)*. The first one enables rules to query the order property, while the latter enables them to update the order state with whatever they do to the tuple stream. This way of doing it integrates nicely with the plan generator [Neu05], and we therefore plan to base our work on this model.

The state machine model mentioned above also has the ability to reason about available groupings. For example, an ordering on (a, b) is automatically a grouping on (a, b) as well, since it can be easily aggregated — but not vice versa. This is useful when optimizing queries with group by-s or projections.

Other Properties

Another interesting property is *sharing* — how much of the plan that is shared (for DAGs), as this alters how the optimizer prunes plans and how the cost calculations are done. First, a plan only dominates another if it is cheaper and offers at least as many sharing opportunities. 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. Also, if a plan that can be shared n times cost more than n times another optimal plan, it can be pruned away. This topic quickly becomes very complicated, so we leave it here for now.

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

5.5.3 Data Structures

Memoization Table

To avoid solving the same subproblem more than once, the plan generator memorizes solutions to problems solved using a hash table. A set of properties uniquely identifies a problem, 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 user in the optimizer, but our testing has discovered that it is in the area of up to 50 MB for moderately sized queries (7-8 relations).

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

```
private Dictionary<BitSet, PlanSet> plansCache;
```

PlanSet

Multiple plans satisfying the same properties are organized in one PlanSet, which contains data common to the plans. As such, a PlanSet can be viewed as a container for the solutions to a subgraph 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,σ}.

PlanSet contains functionality to prune dominated plans as new plans are added to the PlanSet using the *AddPlan* method. Thereby, the set will never contain any plan dominated by any other plan (e.g. in terms of cost and ordering), and hence the optimizer will never store such plans.

Listing 5.5: PlanSet class (simplified)

```
public class PlanSet : IEnumerable<Plan> {
    private List<Plan> plans = new List<Plan>();
    public BitSet Properties { get; set; }
    public IPlanSetState State { get; set; }

    public void AddPlan(Plan planToAdd) {
        // Simplified: Add plan to PlanSet if cheaper than
        // existing plans in the set, removing more expensive plans.
    }
    internal Plan GetCheapest() { // Simplified: Return cheapest plan. }
}
```

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 subplans. It is as compact as possible, as potentially many millions will be created.

Listing 5.6: Plan class (simplified, interface only)

```
public class Plan {
    public PlanSet PlanSet { get; set; }
    public IRule Rule { get; set; }
    public ICost Costs { get; set; }
    // Todo: Ordering, Shared bit sets.
    public List<Plan> Children { get; set; }
    public CostRelation Compare(Plan other);
    public CostRelation CompareTotal(Plan other);
}
```

The *PlanSet* property contains a reference to the enclosing PlanSet, while *Rule* contains the rule that constructed this plan. *Costs* stores this plan's costs. The list *Children* contains all subplans 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. The *Compare* and *CompareTotal* methods enable the optimizer to compare plans based on cost and (in the future) ordering.

As the "todo" in the code snippet suggests, we have not made the optimizer aware of orderings and extended DAG sharing yet. This is planned for the upcoming semester.

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.

Listing 5.7: BasicPlanSetState class (simplified)

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

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. The cardinalites in edge labels on figures in Section 7.3 are actually taken from PlanSetStates for the nodes below them.

Costs

Each plan has an associated cost, stored in the Plan class as an instance of *ICost*. The definition and explanation of ICost can be found in 3.4.1. In short, all the optimizer core cares about is the *Compare* and *CompareTotal* methods.

Listing 5.8: BasicCost class (simplified, interface only)

```
public class BasicCost : ICost {
    public double Cost { get; set; }
    public static BasicCost operator +(BasicCost a, BasicCost b);
    public CostRelation Compare(ICost other);
    public CostRelation CompareTotal(ICost other);
}
```

For our simple cost model, we have implemented the plan cost as a single double floating point number where lower is better. In the future, we plan to improve this to take random and sequential disk reads, CPU and memory usage into consideration. The model is implemented in the class *BasicCost*, containing a property *Cost* - the cost. The + operator is overloaded for convenience while *Compare*/*CompareTotal* implements *ICost*. The costs in edge labels on figures in Section 7.3 are actually taken from BasicCosts for the nodes below them.

5.5.4 Preparation

The preparation phase has several responsibilities:

Instantiate applicable constructive rules. The input query operator graph is analyzed, and any rules found to be relevant will be instantiated. Only relevant rules will be considered to keep the size of the search space down. For example, a JoinRule is instantiated for each join operator found.

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 rule 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 5.2, Table 5.1 shows the instantiated rules and their *Produced* and *Required* properties.

Minimize bit properties. As described in Section 5.5.1, bit properties are minimized during this phase.

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 6.3.1. 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.

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.

5.5.5 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 shown below.

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

When called, it will generate all 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*. 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*. *Infinity* is supplied for *limit*, as we have no limit yet. This would be improved with heuristics — a better initial limit.

The optimizer 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 controls the direction of search and call back to the optimizer, requesting solutions to subproblems (subset of properties) as their input. Basically, `GeneratePlans` invokes the search rules, which again calls `GeneratePlans`. A very simplified version of our simplest search rule, the rule for creating selections, is shown below (the implementation is actually in UnaryRule, which SelectionRule inherits from).

Listing 5.9: SelectionRule.Search(), very simplified (actually UnaryRule.Search())

```
public override void Search(PlanSet plans, ICost limit) {
    foreach (Plan inputPlan in qo.GeneratePlans(plans.Properties - Produced, limit)) {
        Plan selectionPlan = new Plan(inputPlan) { Rule = this };
        plans.AddPlan(selectionPlan);
    }
}
```

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 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 the caller of *SelectionRule.Search()*. Note that any dominated plans are automatically pruned inside the *AddPlan* method, so no plan found to be dominated in terms of cost and ordering 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 *memorizes* solutions to subproblems (actually partial query plans) with the property set produced by the plan as the memoization key. Thus, if the same subproblem (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). This also implicitly gives limited support for DAGs (but more work is required to fully support it), since solutions to subproblems are reused.

Implementation

Listing 5.10 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 solved this problem before, and we just return the PlanSet stored in the entry. Next, a sanity-check is performed on lines 8-10 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. Memorizing 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 14. Then, in the loop on lines 15-22, we try to apply all search rules, but only if it is relevant to the goal (have its requirements satisfied). We call *ISearchRule.Search*, then we get the cheapest plan that was just generated and use it to lower the cost bound used for cost-based pruning. Finally, we store the generated plans in the memoization table and return them.

Listing 5.10: QueryOptimizer.GeneratePlans(), simplified

```
 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         plansCache[goal] = null;
10         return null;
11     }
12
13     // Construct a new PlanSet and apply all applicable rules.
14     plans = new PlanSet() { Properties = goal };
15     foreach (ISearchRule searchRule in searchRules)
16         if (searchRule.IsRelevantTo(goal)) {
17             searchRule.Search(plans, limit);
18             // Try to lower the limit for cost-based pruning
19             Plan cheapest = plans.GetCheapest();
20             if (cheapest != null && cheapest.Costs.Compare(limit) == CostRelation.Better)
21                 limit = cheapest.Costs;
22         }
23
24     plansCache[goal] = plans;
25     return plans;
26 }
```

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 below.

Listing 5.11: GoalIsUnreachable()

```
private bool GoalIsUnreachable ( BitSet goal ) {
    BitSet mask = BitSetManager . Empty;
    foreach ( IProducerRule producerRule in rules )
        if ( producerRule . Filter <= goal )
            mask |= producerRule . Filter ;
    return mask != goal ;
}
```

Sample Search

The top two rules in Table 5.1 are the instantiated base rules for our sample query, while the bottom two are the instantiated search rules. Given that the selection rule is applied first, the plans shown in Figure 5.3 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 subplans.

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 creating plan 4. This again triggers the creation of plan 3 by using the join rule. Then it tries to create the goal by using the join rule, creating plan 6, which triggers the creation of plan 5. 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.

To see the actual result of this query optimized, see Section 7.3.1.

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 5.12. The full implementation can be found in Section 6.3.6. 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 subplan in all possible ways, plans requested from the plan generator and added to the plan set.

Listing 5.12: JoinRule.Search(), very simplified

```
public override void Search( PlanSet plans , ICost limit ) {
    BitSet wantedProperties = plans . Properties − (Produced | RequiredLeft | RequredRight) ;

    foreach ( BitSet left in wantedProperties .Walk() ) {
        BitSet right = wantedProperties − left ;
        foreach ( Plan leftPlan in qo. GeneratePlans ( RequiredLeft | left , limit )
            foreach ( Plan rightPlan in qo. GeneratePlans (RequredRight | right , limit )
                plans . Add(new Plan( leftPlan , rightPlan )) ;
    }
}
```

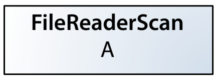
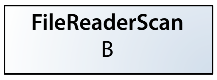
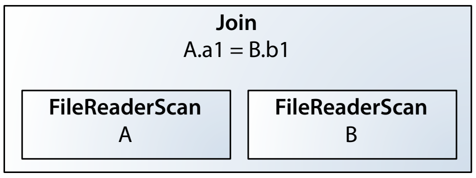
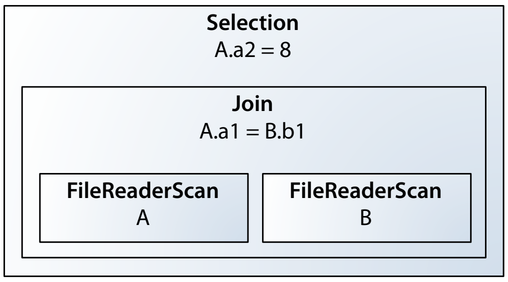
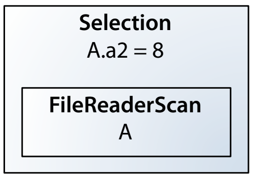
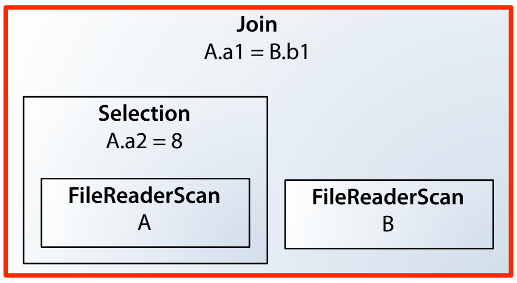
Plan Properties	Plan	Enter Order	Exit Order
a1,a2		0	0
b1,b2		0	0
a1,a1,b1, b2,⋈		2	1
a1,a2,b1, b2,σ,⋈	 Selection A.a2 = 8 Join A.a1 = B.b1 FileReaderScan A FileReaderScan B	1	2
a1,a2,σ		3	3
a1,a2,b1, b2,σ,⋈	 Join A.a1 = B.b1 Selection A.a2 = 8 FileReaderScan A FileReaderScan B	2	4

Figure 5.3: Plans generated for the query in Figure 5.2. The final plan is highlighted.

5.5.6 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 subplans, which again may reference more subplans. 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 optimized query.

Listing 5.13 shows the *BuildAlgebra* method for SelectionRule.

Listing 5.13: SelectionRule.BuildAlgebra(), 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.Children[0].Rule.BuildAlgebra(plan.Children[0]);

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

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

5.6 Graph Pattern Matching

Graph pattern matching is the procedure of finding a subgraph in a graph that matches a given pattern. In our context, it is finding a matching pattern in the query operator DAG. We use this to declare patterns for transformation rules that transform the operator graph, usually by modifying the subgraph that matched the pattern (for example, merging two consecutive projections). 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 6.3.1. Patterns make it easy for implementers of rules to express what they are looking for, instead of having to look for it themselves using code.

Patterns are expressed declaratively 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 defines two lists, *Children* and *Parents*, which make it possible to connect matchers together in a graph, effectively constructing a graph pattern.

Listing 5.14: AbstractNodeMatcher, simplified

```
public abstract class AbstractNodeMatcher {
    public List<AbstractNodeMatcher> Children { get; set; }
    public List<AbstractNodeMatcher> Parents { get; set; }
}
```

We now list the different node matchers prototyped so far:

NodeTypeMatcher matches an operator node of a given type, for example Selection or FileReaderScan.

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

- **SetPreserving** Preserves the set of tuples, i.e. the operator is not allowed to remove or add tuples to the set.
- **DataPreserving** Preserves the data in the tuples, i.e. the operator is not allowed to alter the tuple data.
- **OrderPreserving** Preserves the tuple order, i.e. the operator is not allowed to alter the order of the tuples.

It allows for arbitrary expressions, for example SetPreserving && !OrderPreserving.

ExpressionMatcher allows for specifying an arbitrary expression (Node $\rightarrow$ boolean) for matching a node. LINQ (Language Integrated Query) compiled expressions are used for performance.

DontcareMatcher matches any node.

ZeroOrMore matches zero or more nodes matching a specified AbstractNodeMatcher (property on the ZeroOrMore class).

For example, the pattern

```
NodeTypeMatcher("Projection") →
ZeroOrMore(NodeBehaviorMatcher(OrderPreserving)) →
NodeTypeMatcher("Selection")
```

matches any occurrence of a projection operator followed by zero or more order preserving operators, followed by a selection operator. To make this easy to express in code, we have added some syntactic sugar to the `AbstractNodeMatcher` class. The pattern above can be expressed in code as:

```
new NodeTypeMatcher("Projection").WithChildren(new ZeroOrMode(new
NodeBehaviorMatcher(OperatorBehavior.OrderPreserving)).WithChildren(new
NodeTypeMatcher("Selection)))
```

Much like regular expressions, nodes can be grouped ("tagged") for easier retrieval when a match is found. For example, `new NodeTypeMatcher("Projection").GroupAs("Bing")` will make it possible to get the part of the graph that matched the projection part of the pattern by the name "Bing".

The result of a match is a `PatternMatch` as shown below. *Sources* is a list of the nodes matching the input leaves of the pattern (bottom-most nodes with the root at the top), while *Sinks* is a list of the output leaves (it is a *list* since we can have DAGs matching the pattern). *Groups* is a dictionary that map group names to nodes matching a declared group.

The nodes referenced by the PatternMatch can be directly modified by the rule declaring the pattern. For instance `Match.Groups["Bing"] = null` will make the pattern matcher delete the node, updating parent/child pointers.

Listing 5.15: PatternMatch

```
public class PatternMatch {
    public List<Node> Sources { get; set; }
    public List<Node> Sinks { get; set; }
    public Dictionary<string, PatternGroup> Groups { get; set; }
}
```

The idea behind expressing the patterns declaratively is that it is up to the optimizer what algorithm to use for finding matches. For example, we will probably only implement a naive algorithm, but it might be of interest to implement something better later. For example, [CGK05] describes an algorithm that runs in polynomial time in directed acyclic graphs, while [Gei08] is a full solution for graph rewriting.

Currently, we have only implemented the `NodeTypeMatcher`, which is being used by the rule binders.

6 Rules: Search Space and Pre-/Post Processing

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

– David Wheeler

6.1 Introduction

As introduced in Section 2.4, rule-based optimization is much more extensible, which is why we chose it for our optimizer. 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. This is achieved using *interfaces* in C#. For example, all transformation rules implement `ITransformationRule`.

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 ruin 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 startup, using reflection, all types (classes) in all known assemblies (dll, .NET equivalent of Java JARs) are enumerated. Those identified as rules are loaded into the optimizer and prepared for optimizer use.

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 `[TransformationRule]` before the class declaration. Constructive rules use a rule binder concept as explained in Section 6.3.1. 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.

6.2 Transformation Rules

6.2.1 Rule Interface

All transformative rules, both pre- and post-processor rules, implement **ITransformationRule**, as shown in Listing 6.1. The *Pattern* property enables the rule to declare the graph pattern that should match a sub graph of the operator graph for this rule to fire. *DependsOn* lists the names of all transformative rules that should be invoked before this one, if more rules are applicable. *Fire* is called with the matching pattern when the rule is invoked. Finally, if this rule should be applied iteratively, that is, applied multiple times until the result stabilizes, it should return true for *Iterative*.

Listing 6.1: ITransformationRule interface

```
interface ITransformationRule {
    AbstractNodeMatcher Pattern { get; }
    List<string> DependsOn { get; }
    void Fire(PatternMatch match);
    bool Iterative { get; }
}
```

Additionally, each transformation rule needs to be tagged with the *TransformationRule* attribute to announce its existence to the optimizer. When detecting this attribute, the optimizer will load the rule and invoke it when its pattern matches. This attribute also decides if the rule is a pre- or post-processor (or both) by taking the type as an argument. To tag a rule as a pre-processor, one would insert this line just before the class declaration: `[TransformationRule(TransformationType.Pre)]`.

6.2.2 Merge Trim and Sort

So far, we have implemented one transformation rule, namely the *MergeTrimSort* rule. This rule comes from the observation that MARS' *Sort* operator includes a *Trim sort option*, optimizing the internal search algorithm to only output a window of n tuples, with an offset o from the top. The alternative is to have a regular sort operator with a *Trim* operator just above it, however, this would not allow the sort algorithm to be optimized for the trim operation.

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.

The implementation of the MergeTrimSort rule is given in Listing 6.2. Most protrusive is the pattern declaration, defining the pattern we are looking for in the operator graph. It uses our declarative pattern language to express what we just explained above. First, it looks for a node with type "TrimOperator". It is grouped as "trim" (just as in regular expressions) to be easy to get back to later. Then, it declares that it should have a single child (the *WithChildren* takes a list of children). This child should be zero or more nodes (below each other) which should be both *SetPreserving* and *OrderPreserving* as discussed above. Further, all of them should only have one parent, satisfying the third condition above. Finally, the last of these zero or more nodes should be a sort operator, grouped as "sort".

Listing 6.2: MergeTrimSort rule (simplified)

```
[ TransformationRule ( TransformationType . Pre  |  TransformationType . Post ) ]
public  class  MergeTrimSort  :  ITransformativeRule  {
    public  override  AbstractNodeMatcher  Pattern  {
        get  {
            return  (new NodeTypeMatcher("TrimOperator"))
                         .GroupAs("trim")
                         .WithChildren(
                             new ZeroOrMore(
                                 NodeBehaviourMatcher.All ( OperatorBehaviour .  SetPreserving
                                     |  OperatorBehaviour . OrderPreserving )
                                 .WithAnyOneParent() // Do not match a  branching  node.
                             )
                             .WithChildren(
                                 (new NodeTypeMatcher("SortOperator")).GroupAs("sort")
                             )
                         );
        }
    }
    public  override  void  Fire ( PatternMatch  match) {
        Node sortOperator  = match.Groups["sort"].OnlyMatch;
        match.Groups["trim"].OnlyMatch = null ;
        sortOperator ["offset"]  =  ...;
        sortOperator ["hitcount"]  =  ...;
    }

    public  override  bool  Iterative  { get  {  return  true ;  } }
}
```

The *Fire* method is called by the optimizer when a match is found, with the matched expression as argument. It sets the trim operator to *null*, indicating that it should be removed, and also sets the properties on the sort operator. *true* is returned for *Iterative* since there could be more than one trim to be merged into the sort. Each rule appliance would merge one occurrence.

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 it here for brevity, but the full code can be found in Appendix A.4.

In the future we plan to implement another rule that detects trim or sort operators with a trim window size set to 0. Such an operator could be replace with a no-op operator producing no tuples.

6.2.3 More 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, 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. 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 subqueries 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 pushdown vs. pull up vs. elimination. Pushdown: 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 subqueries. In some cases, multiple subqueries can be merged to a single subquery. For example, if multiple subqueries fetch data from the same table, a merge may be possible.

Inline functions. It may be beneficial to inline functions, i.e. make subqueries of them.

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

6.3 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 know that adding multiple alternative rules for the same query parts will increase the size of the search space accordingly. 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

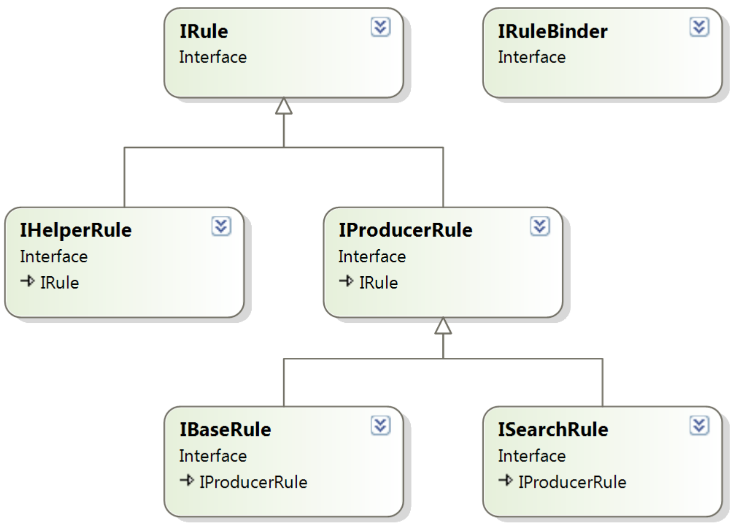


Figure 6.1: Class diagram for the constructive rule interface hierarchy.

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, before we explain the different rules we have implemented. At this stage, we have implemented rules for file reader scans (analogous to table scans), selections and joins. This enables us to do some selection predicate move-around and join ordering. This is demonstrated in Section 7.3.

6.3.1 Rule Interface

The borderline between the optimizer core and the different search rules includes multiple interfaces in a hierarchy, as shown in Figure 6.1. 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, shown in Listing 6.3 is the base interface for all constructive rules. The *Name* property returns the name of the rule for display- and debug purposes. The *UpdatePlan* method is used during plan generation and updates a given plan's cost and rule instance. Finally, the *BuildAlgebra* method is used during the reconstruction phase when each rule recursively constructs the final operator graph.

Listing 6.3: IRule interface

```
public interface IRule {
    string Name { get; }
   void UpdatePlan(Plan plan);
   Node BuildAlgebra(Plan plan);
}
```

Helper rules, which implement *IHelperRule*, as shown in Listing 6.4 are rules that the optimizer does not directly know or reason about. Helper rules are typically used by other search rules. For instance, the Join rule uses the HybridHashJoinRule or (in the future) 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.

Listing 6.4: IHelperRule interface

```
public interface IHelperRule : IRule { }
```

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. *IProducerRule*, as shown in Listing 6.5 contains the following members. The *Produced* property gives which properties this rule can be used to achieve. *Filter* is used by the query optimizer to filter out unapplicable rules and unreachable plans, as described in Section 5.5.5. Each search rule is given an *Id* to identify it to the optimizer, and the *Node* property points to the original node in the input operator graph it was instantiated from.

Listing 6.5: IProducerRule interface

```
public interface IProducerRule : IRule {
    BitSet Produced { get; set; }
    BitSet Filter { get; }
    int Id { get; }
    Node Node { get; set; }
}
```

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 FileReader-ScanRule. The *Initialize* method, as shown in Listing 6.6, is called when the rule is initialized during the preparation phase of plan generation.

Listing 6.6: IBaseRule interface

```
public interface IBaseRule : IProducerRule {
    void Initialize (PlanSet plans);
}
```

Search rules are the most important rules. They both produce and require bit properties, and model the internal nodes in the operator graph. Examples include *Selection* and *Join*. More important, 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 can be 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.

Listing 6.7: ISearchRule interface

```
public interface ISearchRule : IProducerRule {
    bool IsRelevantTo (BitSet goal);
    void Search(PlanSet planSet, ICost limit);
    IList<BitSet> Required { get; }
}
```

Finally, **rule binders** provides the optimizer with a way to instantiate relevant rules for the query to be optimized. The basic functionality of a rule binder is to declare a *Pattern* that matches nodes in the input operator graph (for instance nodes of type SelectOperator) and instantiate and return rules with properties set. For instance, the SelectionOperatorBinder sets the selectivity of the selection.

Classes that implement *IRuleBinder*, and are tagged with the `[RuleBinder]` attribute will be loaded by the optimizer upon startup. For each query to be optimized, all binders having matching patterns will be invoked. After assigning itself to the *QueryOptimizer* property,

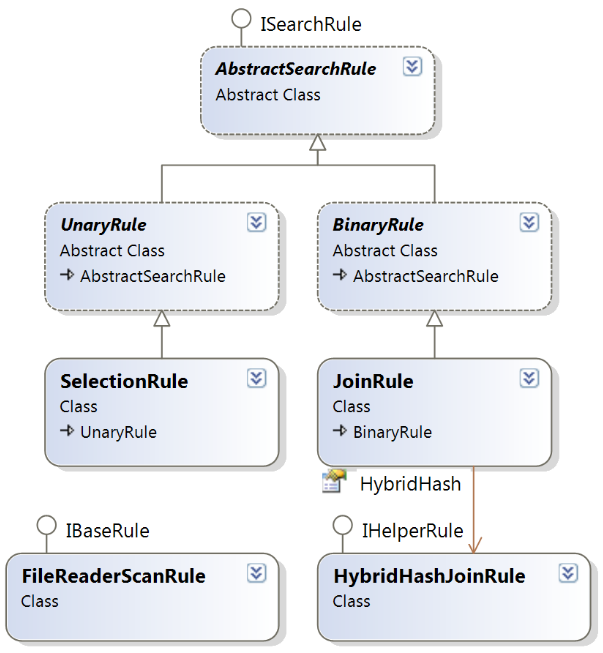


Figure 6.2: Class diagram for the implemented search rules.

the query optimizer will invoke *InitializeBitSets* with the matches in the operator graph. This method is supposed to add the names of the required and produced bit properties to the BitSetManager. Then *InitializeRules* is called. It should instantiate rules and set properties on them, like the Produced and Required properties. Finally, the optimizer calls *GetRules* to fetch the instantiated rules.

Listing 6.8: IRuleBinder interface

```
public interface IRuleBinder {
    AbstractNodeMatcher Pattern { get; }
    void InitializeBitSets (IEnumerable<PatternMatch> matches);
    void InitializeRules ();
    IEnumerable<IProducerRule> GetRules();
    QueryOptimizer QueryOptimizer { get; set; }
}
```

For a complete example of a rule binder, see Listing A.4 in Section A.3.

6.3.2 Scan Rules

We now continue by describing the three (or four, including helper rules) rules we have implemented. Their relationship can be seen in Figure 6.2. *AbstractSearchRule*, *UnaryRule* and *BinaryRule* are abstract rules implementing common functionality. First out are the most basic rules, the general class of scan rules.

Scan rules model the different access paths the system supports, the most basic being table scans, or in our case, file reader scans since this is what is used in our copy of MARS. Index scans and index lookups are also modeled as base rules, the primary difference being that they produce ordered output. Index lookups also applies a selection predicate and are modeled as a base rule that also produces the bit property "selection applied".

The *FileReaderScanRule* is presented, somewhat simplified, in Listing 6.9. The *Node* property points to the node the rule was instantiated from, while *Cardinality* is the estimated cardinality for this scan (taken from statistics). For this rule, *Filter* is equal to *Produced*, and contains the attributes produced by this scan. *Id* is the unique id for this rule in this optimization and *Name* is a user-friendly name.

The *Initialize* method, is called when the rule is initialized during the preparation phase of plan generation and sets some basic data read from system catalogs and statistics in the PlanSet-State. *UpdatePlan* will also be called during preparation and updates a given plan's cost. Finally, the *BuildAlgebra* method is used during the reconstruction phase and creates a new node and copies the properties from the original node in the input query. This will be refined when it is integrated closed with MARS.

Listing 6.9: FileReaderScanRule class (simplified)

```
public class FileReaderScanRule : IBaseRule {
    public Node Node { get; set; }
    public double Cardinality { get; set; }
    public BitSet Filter { get; set; }
    public BitSet Produced { get; set; }
    public int Id { get; private set; }
    public string Name { get { return "FileReaderScan"; } }

    public void Initialize (PlanSet plans) {
        plans.State = new BasicPlanSetState () { Cardinality = ..., TupleSize = ... };
    }

    public void UpdatePlan(Plan plan) {
        plan.Costs = new BasicCost( Cardinality * 0.001);
    }

    public Node BuildAlgebra(Plan plan) {
        Node newNode;
        if (queryOptimizer.ReconstructionTable.TryGetValue(plan, out newNode))
            return newNode;

        newNode = new Node() { OperatorType = Node.OperatorType };
        newNode.Properties = Node.Properties;

        queryOptimizer.ReconstructionTable[plan] = newNode;
        return newNode;
    }
}
```

6.3.3 Unary Rule

Unary rules are rules with only one child, and most of them can be implemented quite easily if one do not want to anything advanced. 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 may make the search space too big (it increases exponentially with the number of rules), so care should be used.

Listing 6.10 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, effectively placing itself on the top. If no plans can be generated, nothing is done. Else, if this is the first plan being produced for these properties, we set some PlanSetState. The output cardinality is input cardinality times selectivity, while tuple size is the same. Finally, for each input plan, a new output plan is created with the input plan as its only child, the plan is updated (with costs etc.) and added to the output.

Listing 6.10: UnaryRule (simplified)

```
public abstract class UnaryRule : AbstractSearchRule {
    public override void Search(PlanSet plans, ICost limit) {
        PlanSet input = qo.GeneratePlans(plans.Properties - Produced, limit);
        if (input == null)
            return;

        if (plans.Count == 0) { // First run, so set some state.
            plans.State = new BasicPlanSetState() {
                Cardinality = input.State.Cardinality * Selectivity,
                TupleSize = input.State.TupleSize
            };
        }

        foreach (Plan inputPlan in input) {
            Plan newPlan = new Plan();
            newPlan.Children = new List<Plan> { inputPlan };
            UpdatePlan(newPlan);
            plans.AddPlan(newPlan);
        }
    }
}
```

6.3.4 Selection Rule

SelectionRule constructs selections in the query. It is a unary rule and therefore inherits from *UnaryRule*. At this stage, we have not optimized its implementation and just left the standard unary search method in place. In the future, we plan to make it smarter to decrease the size of the search space. One way to do it would be to pre-compute its position using heuristics, using operator dependencies in the bit properties to force it to stay in one location. Another way is to a make it smarter than just putting itself everywhere.

See *UnaryRule.Search* in Listing 6.10 for the *Search* method and Listing 5.13 in Section 5.5.5 (Reconstruction Phase) for the *BuildAlgebra* method. Listing 6.11 shows the rest of the implementation, the *UpdatePlan* method which mainly updates the plan costs. The plan cost is the input cost plus the cost of evaluating the predicate for each tuple.

Listing 6.11: SelectionRule (simplified)

```
public class SelectionRule : UnaryRule {
    public double PredicateCost { get; set; }
    public override void UpdatePlan(Plan plan) {
        plan.Rule = this;
        CalculateCosts(plan);
    }
    private void CalculateCosts(Plan plan) {
        BasicCost childCost = plan.Children[0].Costs;
        BasicPlanSetState childState = plan.Children[0].PlanSet.State;
        plan.Costs = childCost + new BasicCost(childState.Cardinality * PredicateCost);
    }
}
```

6.3.5 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, but may be extended with more functionality in the future.

Listing 6.12: BinaryRule

```
public abstract class BinaryRule : AbstractSearchRule {
    public BitSet RequiredLeft { get { return Required[0]; } }
    public BitSet RequiredRight { get { return Required[1]; } }
}
```

6.3.6 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. We gave a more simplified code sample in the end of Section 5.5.5; it might be wise to start there if not already read.

[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 9 in Listing 6.13 checks which one is set. Let us consider bushy join trees 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 *RequiredLeft/RequiredRight* properties must be requested from the left or right one (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 *wantedProperties = Requested properties - (Produced | RequiredLeft | RequiredRight)* and happens on line 10. Then *InternalSearch* is called to produce plans.

For left-deep enumeration, the difference is how the freely chosen properties are determined. When the join rules are initialized, a property *OtherJoinRules* 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, the freely chosen properties, *wantedProperties = Requested properties - (Produced | OtherJoinRules)* on line 14. *remainingJoins* is computed as the joins remaining below this rule. The ones performed above this one, closer to the root, are removed. All remaining joins are explicitly forced to the left input by specifying *RequiredLeft | remainingJoins* as the left parameter to *InternalSearch*. *InternalSearch* needs to be called a second time with *RequiredLeft/RequiredRight* swapped (but with remainingJoins still on the left side) to enable join ordering at all (otherwise it would just force one possible order.

Listing 6.13: JoinRule, simplified

```
1  public class JoinRule : BinaryRule {
2      public HybridHashJoinRule HybridHash { get; set; }
3      public BitSet OtherJoinRules { get; set; }
4      public JoinEnumeration JoinEnumeration { get; set; }
5      public override string Name { get { return "Join"; } }
6
7      public override void Search(PlanSet plans, ICost limit) {
8          BitSet wantedProperties, remainingJoins;
9          if (JoinEnumeration == JoinEnumeration.Bushy) {
10             wantedProperties = plans.Properties − (Produced | RequiredLeft | RequiredRight);

11             InternalSearch(plans, limit, ref wantedProperties, RequiredLeft, RequiredRight);
12         }
13         else if (JoinEnumeration == JoinEnumeration.LeftDeep) {
14             wantedProperties = plans.Properties − (Filter | OtherJoinRules);
15             remainingJoins = (OtherJoinRules & plans.Properties) − Filter;
16                 InternalSearch (plans, limit, wantedProperties, RequiredLeft | remainingJoins,
                       RequiredRight);
17                 InternalSearch (plans, limit, wantedProperties, RequiredRight | remainingJoins,
                       RequiredLeft);
18             }
19         }
```

Then the freely chosen properties are distributed between the left and right subplan 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 21. 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 *rightProperties* on line 26 and generate the right input plans. If the PlanSet is empty, we then initialize its PlanSetState on line 31. Finally, we create new plans for each possible input plan we found on lines 38-45.

The *JoinRule* itself only represents the *logical* join. The different *physical* join algorithms are represented by helper rules. In the loops in lines 38-45, one plan is actually added for each join algorithm (in the future). 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*.

Listing 6.14: JoinRule, simplified

```
20     private void InternalSearch (PlanSet planset, ICost limit, BitSet wantedProperties,
           BitSet left, BitSet right) {
21         foreach (BitSet leftProperties in wantedProperties.Walk()) {
22             PlanSet leftPlans = queryOptimizer.GeneratePlans(left | leftProperties, limit);
23             if (leftPlans == null || leftPlans.GetCheapest().Costs.Compare(limit) ==
                   CostRelation.Worse)
24                 continue;
25 
26             BitSet rightProperties = wantedProperties − leftProperties;
27             PlanSet rightPlans = queryOptimizer.GeneratePlans(right | rightProperties, limit
                   );
28             if (rightPlans == null)
29                 continue;
30 
31             if (plans.Count == 0) { // First plan, so set some state.
32                 plans.State = new BasicPlanSetState () {
33                     Cardinality = leftPlans.State.Cardinality * rightPlans.State.Cardinality
                           * Selectivity,
34                     TupleSize = leftPlans.State.TupleSize + rightPlans.State.TupleSize
35                 };
36             }
37 
38             foreach (Plan leftPlan in leftPlans) {
39                 foreach (Plan rightPlan in rightPlans) {
40                     Plan plan = new Plan(leftPlan, rightPlan);
41                     HybridHash.UpdatePlan(plan);
42                     // Todo: Consider MergeJoin, NestedLoopsJoin.
43                     plans.AddPlan(plan);
44                 }
45             }
46         }
47     }
```

The logical join rule should never build any algebras (this is left up to the helper rules), so *BuildAlgebra* just throws an exception. *UpdatePlan* should never be called either, but if it is, we just delegate it to the HybridHashJoin rule.

Listing 6.15: JoinRule, simplified

```
    public override void UpdatePlan(Plan plan) {
        HybridHash.UpdatePlan(plan);
    }
    public override Node BuildAlgebra (Plan plan) {
        throw new NotImplementedException();
    }
}
```

HybridHashJoin Rule

The *HybridHashJoinRule* is a helper rule, and not directly used by the optimizer, but by the JoinRule. Therefore, it does not implement the *Search* method, but only *UpdatePlan* and *BuildAlgebra*. *UpdatePlan* updates the cost of the plan after the formula $\mathcal{C}(\mathsf{A} \bowtie \mathsf{B}) = \mathcal{C}(\mathsf{A}) + \mathcal{C}(\mathsf{B}) + |\mathsf{A} \bowtie \mathsf{B}|$, that is, the sum of the children costs plus the cardinality of the join. *BuildAlgebra* constructs an operator node with the input plans as children. As of now, it just copies the properties and node time from the input query, but this will be changed to actually instantiate a join operator of the correct type when we integrate with MARS.

Listing 6.16: HybridHashJoinRule, simplified

```
public class HybridHashJoinRule : IHelperRule {
    private JoinRule parent;
    public string Name{ get { return "HybridHashJoin"; } }
    public void UpdatePlan(Plan plan) {
        plan.Rule = this;
        CalculateCosts (plan);
    }
    private void CalculateCosts (Plan plan) {
        Plan leftInput = plan.Children [0], rightInput = plan.Children [1];
        plan.Costs = leftInput .Costs + rightInput .Costs + new BasicCost(( leftInput . State .
            Cardinality *
            rightInput . State . Cardinality * parent . Selectivity ));
    }
    public Node BuildAlgebra (Plan plan) {
        Node newNode;
        if (queryOptimizer. ReconstructionTable .TryGetValue(plan, out newNode))
            return newNode;

        Node inputLeft = plan.Children [0]. Rule. BuildAlgebra (plan.Children [0]);
        Node inputRight = plan.Children [1]. Rule. BuildAlgebra (plan.Children [1]);

        newNode = new Node() { OperatorType = parent .Node.OperatorType };
        newNode.Children.Add( inputLeft );
        newNode.Children.Add(inputRight);
        newNode. Properties = parent .Node. Properties ;

        queryOptimizer. ReconstructionTable [plan] = newNode;
        return newNode;
    }
}
```

7 Current State

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. Finally, we present a series of sample runs of the optimizer.

7.1 Results

In summary, we have implemented a running optimizer. As of now, it does not consider useful orderings, the cost model is fairly simple and it only supports three operators. Nevertheless, it is running and able to produce optimal plans (as shown in this chapter) and tackles the important problem of join ordering.

It is also based on an architecture which addresses the design goals introduced in Section 5.1. It is *extensible* and the rule architecture enables support for arbitrary cost models and pre- and post-processing, while the cost model is external to the optimizer itself. The design is documented in this report and exploits the object-oriented features of C#. Further, the design supports what we plan to implement in the future, like DAG-structured query plans, useful orderings and support for arbitrary operators. 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 7.2. Performance is a challenge to all query optimizers because the general problem is exponential in nature. We plan to introduce heuristics to address the problem.

We believe we have created a solid foundation for the further work in our master thesis next semester.

7.1.1 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 if not putting any restrictions on the search space. 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.

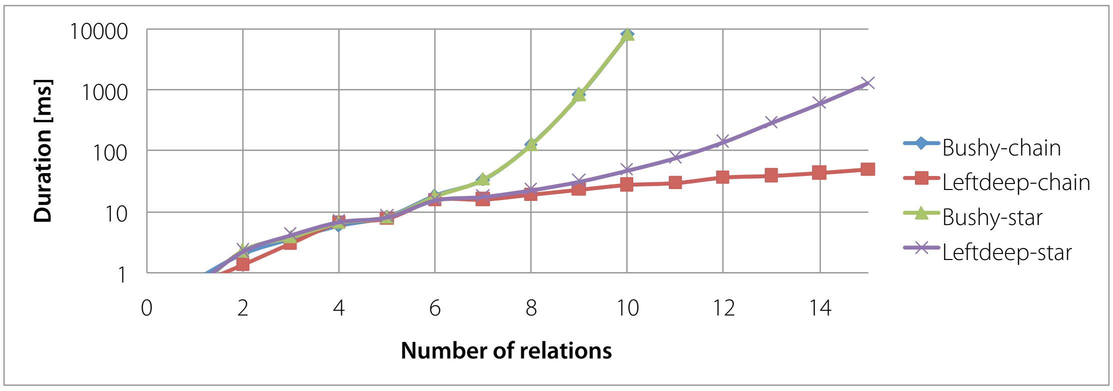


Figure 7.1: Plan generation for chain and star queries, bushy and left-deep, average of 100 runs.

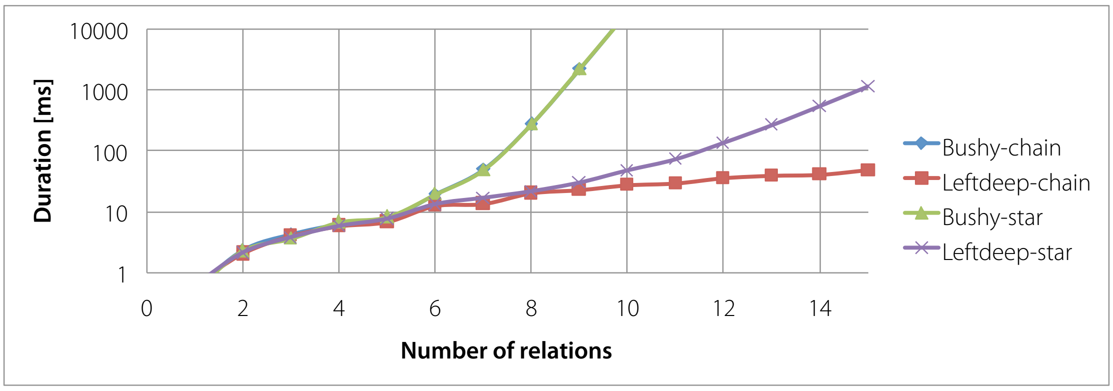


Figure 7.2: Same as Figure 7.1, without reachability caching.

We cannot give a complete picture of the performance of our optimizer since it only supports joins and selections at this stage, but nevertheless, we have done some performance testing on raw join reordering. Figure 7.1 shows time spent optimizing plotted against the number of relations. *Star queries* are queries where all relations 1..n are joined to relation 0, while in *chain queries*, relations are joined in a chain, ie. 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.

The plot clearly shows that bushy plans have higher complexity and take longer to optimize than left-deep plans. Further, it also shows that star queries are more complex than chain queries, as expected. We consider anything close to or above one second to be too long, which suggests that, at its current state, enumerating bushy plans should be abandoned for > 8 relations (possibly lower). Enumerating left-deep plans is feasible up to around 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. We can see that we have a "bump" in the graph around 5 relations. We cannot explain what this is, but it may be an artifact from garbage collection or memory/cache locality.

Figure 7.2 shows the same timings, but now with reachability caching (as explained in the list in Section 5.5) disabled. We can see that this results in better run times for bushy enumeration, especially for more than 7 relations. Performance for left-deep plans is marginally decreased (by 50-100ms), but this is because left-deep enumeration does not generate as many unreachable plans as bushy enumeration does.

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 7.3. 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 the same exponential increase in both graphs, but ours come two relations "earlier" and is more exponential.

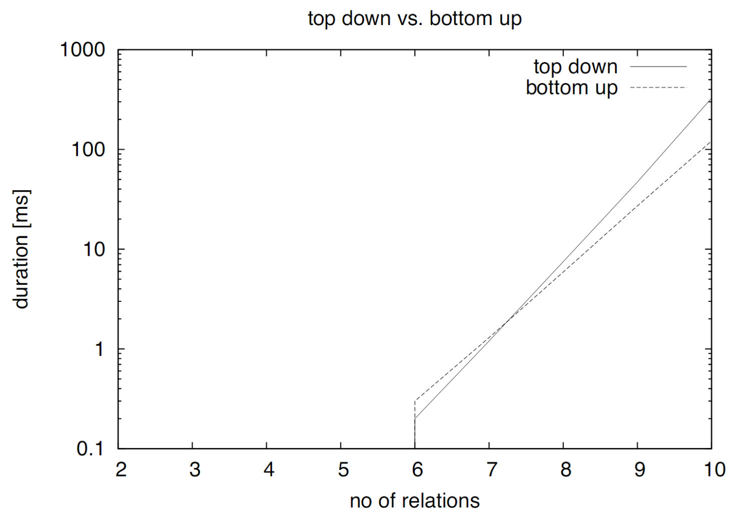


Figure 7.3: Plan generation performance as reported in [Neu05].

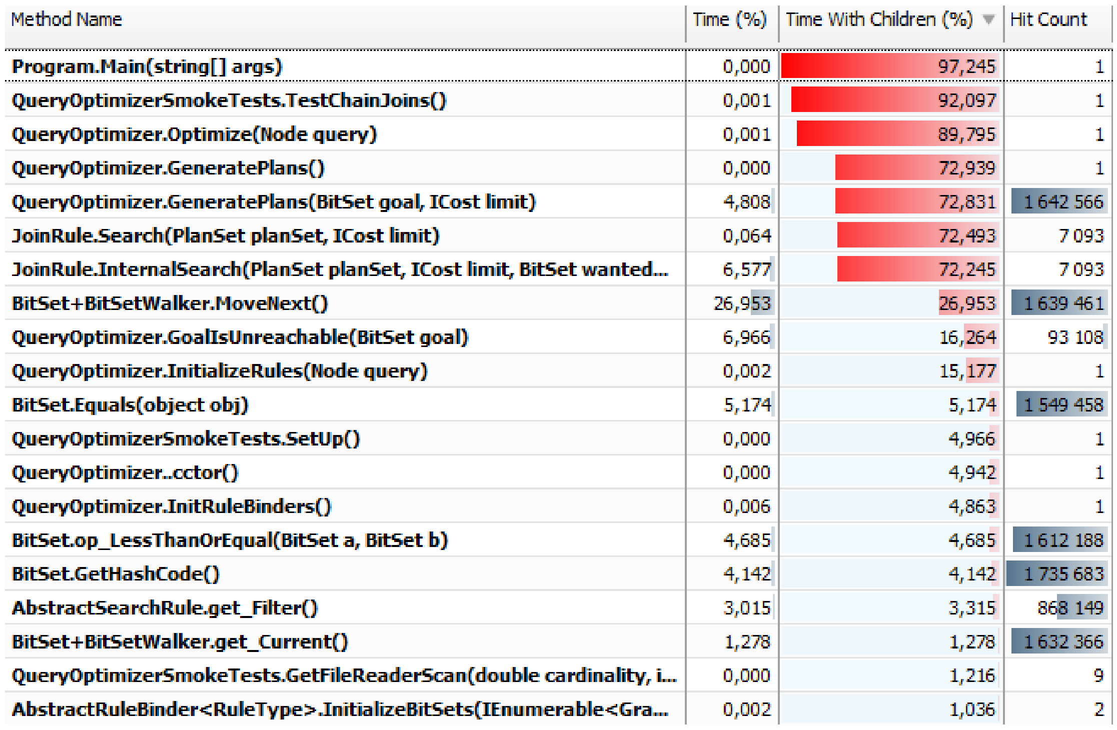
This does not look too good on our part, but we believe there are multiple reasons. First, Neumann's implementation is probably more optimized and refined than ours. He may also do some logical optimizations (as opposed to just code optimizations) we do not. We have not spent much time optimizing our solution, as we wanted to get the design principles right first. Second, we only do basic cost-based pruning, and can probably improve our solution here. Addressing these points will be one of the tasks for the upcoming semester. See Section 7.2.1 for details.

Our performance tests were conducted on a 2.0 GHz Intel Core Duo (but the optimizer is single threaded, so far) on .NET Framework 3.5 SP1 on Windows Vista. We have not measured memory usage accurately, but by inspecting the memory use of the optimizer process while running, we see approximately 50 MB for the largest queries (9-15 relations), less for the smaller.

We have not integrated our optimizer with MARS at this stage so we have no execution statistics to present. Still, we feel confident that queries optimized by our optimizer will be able to execute faster than non-optimized queries. Execution performance will be evaluated in the upcoming semester.

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 7.4 shows a sample run using the ANTS Profiler [Red08] when performing join ordering of 9 relations.



Method Name	Time (%)	Time With Children (%) ▼	Hit Count
Program.Main(string[] args)	0,000	97,245	1
QueryOptimizerSmokeTests.TestChainJoins()	0,001	92,097	1
QueryOptimizer.Optimize(Node query)	0,001	89,795	1
QueryOptimizer.GeneratePlans()	0,000	72,939	1
QueryOptimizer.GeneratePlans(BitSet goal, ICost limit)	4,808	72,831	1 642 566
JoinRule.Search(PlanSet planSet, ICost limit)	0,064	72,493	7 093
JoinRule.InternalSearch(PlanSet planSet, ICost limit, BitSet wanted...	6,577	72,245	7 093
BitSet+BitSetWalker.MoveNext()	26,953	26,953	1 639 461
QueryOptimizer.GoalIsUnreachable(BitSet goal)	6,966	16,264	93 108
QueryOptimizer.InitializeRules(Node query)	0,002	15,177	1
BitSet.Equals(object obj)	5,174	5,174	1 549 458
QueryOptimizerSmokeTests.SetUp()	0,000	4,966	1
QueryOptimizer..cctor()	0,000	4,942	1
QueryOptimizer.InitRuleBinders()	0,006	4,863	1
BitSet.op_LessThanOrEqual(BitSet a, BitSet b)	4,685	4,685	1 612 188
BitSet.GetHashCode()	4,142	4,142	1 735 683
AbstractSearchRule.get_Filter()	3,015	3,315	868 149
BitSet+BitSetWalker.get_Current()	1,278	1,278	1 632 366
QueryOptimizerSmokeTests.GetFileReaderScan(double cardinality, i...	0,000	1,216	9
AbstractRuleBinder<RuleType>.InitializeBitSets(IEnumerable<Gra...	0,002	1,036	2

Figure 7.4: Profiler run of a 9 relation chain join query.

The figure lists the methods where the most time was spent during execution, along with their hit counts. The first four lines is just 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 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. We plan to use profiling as a tool continuously through or project to make sure the code we write performs well.

7.2 Identified Issues and Suggested Solutions

7.2.1 Exhaustive Enumeration

Currently, our optimizer is close to exhaustive within the limitations set on the search space and 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$12 for left-deep plans (where too slow is $\geq$200ms).

According to our *fast*-representative, more than twelve relations are not too common in MARS queries. Nevertheless, we want to look into ways of addressing this problem.

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 subproblem. Whenever a new exploration task exceeds this bound, it can not possible yield a better plan and is aborted.

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.

7.2.2 Predicate Splitting

Currently, a selection operator with its entire predicate is treated as a single unit. This becomes a problem when the predicate should be split and moved to different parts of the operator graph. For instance, consider the query in Figure 7.5a. If the selection expression was `Id1 < 50 && Name2 = "Foo"`, a likely optimization would be to split it (it can be splitted directly since it is in conjunctive normal form) and push each predicate down through the join, right after the file scans in both inputs.

In our current implementation, the optimizer will not be able to do this, since we have not implemented expression parsing and splitting.

Suggested Solution

The first thing that needs to be done is to parse the selection expression and split it. MARS has routines for parsing such expressions, which we plan to leverage. It may also be necessary to simplify it or apply De Morgan's laws multiple times before splitting it. This can be done during the pre-processing stage of optimization. Then the selection operator would be split into multiple consecutive operators, which can be freely moved around (given that the query semantics are not changed). However, this could lead to an increase in search space size, so we may want to make the selection rule aware of the possibility to split the predicate in some other way.

We still need to determine the best approach, but it certainly possible to solve.

7.2.3 Unreachable Plans

Consider the query in Figure 7.6, which joins four relations. Following the process described in 5.5.4, 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* = {Id2}, *RequiredRight* = {Id3}.

Following the algorithm for the join rule, described in 6.3.6, the rule will try to satisfy *Goal* by exhaustively splitting *wanterProperties* $=$ *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 plan to pre-compute transitive closures on the required properties of all rules instantiated during the preparation phase. This will 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 rule would consult this array before iterating over all the different ways of splitting the *wantedProperties* bitset.

7.3 Sample Query Optimizations

To show that we have a running query optimizer, we include some sample query optimizations of a some selected queries that illustrate a few of the optimizer's current capabilities. For most samples, we include a before and after operator graph.

Note that all figures used in this section have been auto-generated by the query optimizer on the fly — nothing is "by hand". To do this, we use a graph visualization software called *Graphviz* [AT 08]. In short, we export the optimizer's internal node structure to the `dot` language (a sample can be found in Section A.2). Then we call the dot tool, which generates a png image (or any other format). Although the graphs below only includes plan costs, cardinalities and a few other things, they can also be made to include attributes, operator properties and any other details. Being able to easily visualize the plans makes it substantially easier to interpret the changes done by optimizer.

7.3.1 Select Through Join

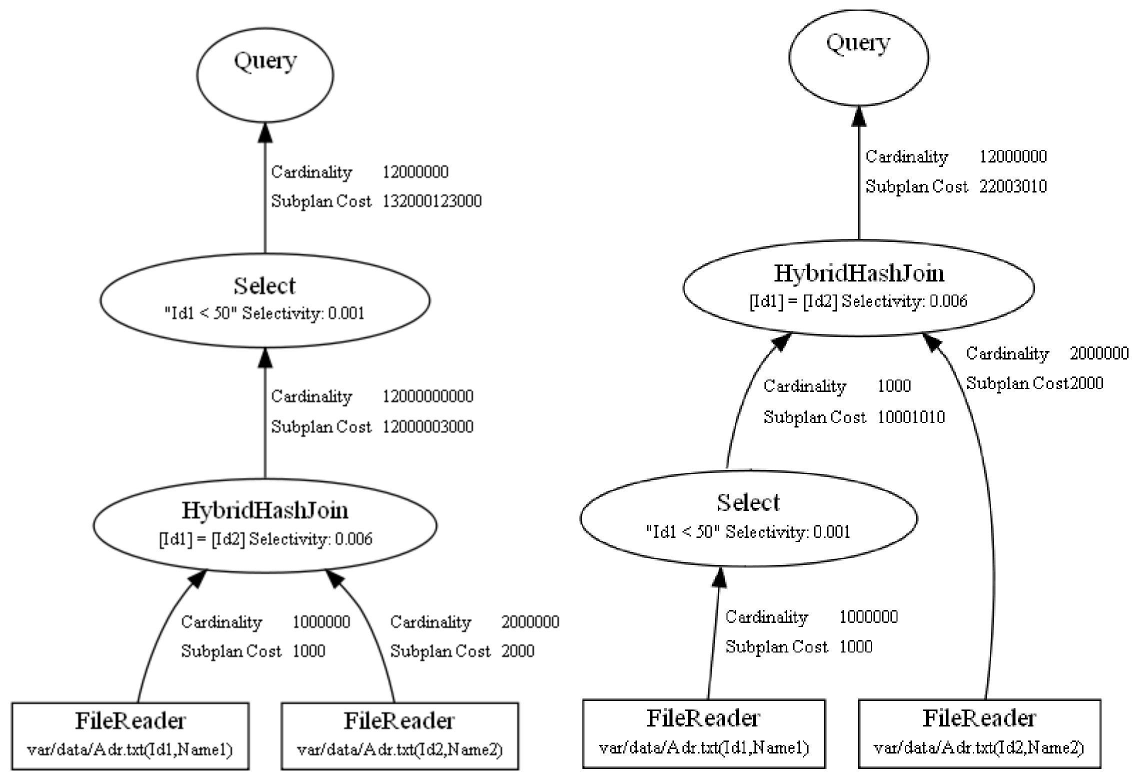
The first sample is a simple SPJ-query (select, project, join), involving two relations and one selection. The two relations are joined on their id attributes and then filtered on `Id1 < 50`. This would roughly correspond to the following SQL query.

```
SELECT * FROM R1
    JOIN R2 ON R1.Id1 = R2.Id2
    WHERE Id1 < 50
```

As we can see on the before graph in Figure 7.5a, the selection is set up to be performed after the join. In general, it is a good idea to push such selections through and before any joins to limit the tuple count as early as possible. If we had an index on the filtered attribute, we could use it to further lower the cost of the query.

Figure 7.5b shows that the optimizer chose to move the selection before the join, since this lowered the input cardinality for the join, thereby lowering the cost of the join and the entire query. However, it is not always possible or advantageous to do so. First, if the filter expression of the select operator is referencing attributes from both input relations, it can be harder to do. An expression like `Id1 < 50 && Name2 = 'Bob'` would be fine, as we could split it and push one part down each join input. But expressions like `Id1 < 50 || Name2 = 'Bob'` or `Id1 < Id2` would be harder, since they cannot be split and would have to be evaluated after the join.

Second, if the join operator is not a hash join, but a nested loop join, we may want to have the join utilize any appropriate index on the relation instead of having the selection do it.



(a) Before optimization (b) After optimization

Figure 7.5: Pushing selection through join.

Third, if the selection predicate is expensive to evaluate (for instance an expensive function call) and not very selective, it may be more cost effective to evaluate it after the join, especially if the join is estimated to be very selective. We have chosen not to show this, as the after graph would be equal to the before graph.

This query was optimized in 2.6 ms.

7.3.2 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. We only have file readers available to us, but the query would roughly correspond to the following SQL query. See Section A.6 for the automated optimizer test that was used to generate the following figures.

```
SELECT * FROM R0
        JOIN R1 ON R0.Id0 = R1.Id1
        JOIN R2 ON R1.Id1 = R2.Id2
        JOIN R3 ON R2.Id2 = R3.Id3
```

The output cardinality for a join $\mathsf{A} \bowtie \mathsf{B}$ is given by $|\mathsf{A}| \times |\mathsf{B}| \times \mathcal{S}(\mathsf{A}, \mathsf{B})$, where $\mathcal{S}(\mathsf{A}, \mathsf{B})$ is the *selectivity* of the join. The *cost* $\mathcal{C}(\mathsf{A} \bowtie \mathsf{B})$ for a join is given by $\mathcal{C}(\mathsf{A}) + \mathcal{C}(\mathsf{B}) + |\mathsf{A} \bowtie \mathsf{B}|$, that is, the sum of the children costs plus the cardinality of the join.

The cardinalities of the input relations are $|\mathsf{R0}| = 10$, $|\mathsf{R1}| = 20$, $|\mathsf{R2}| = 20$, $|\mathsf{R3}| = 10$. The absolute values are not interesting, but the ratios are — the numbers could be in million rows. The selectivities of the joins are $\mathcal{S}(\mathsf{R0}, \mathsf{R1}) = 0.01$, $\mathcal{S}(\mathsf{R1}, \mathsf{R2}) = 0.5$, $\mathcal{S}(\mathsf{R2}, \mathsf{R3}) = 0.01$, while the costs of the FileReaderScans are $|\mathsf{R}| \times 0.01$ each (low, so we can focus on the join ordering for now).

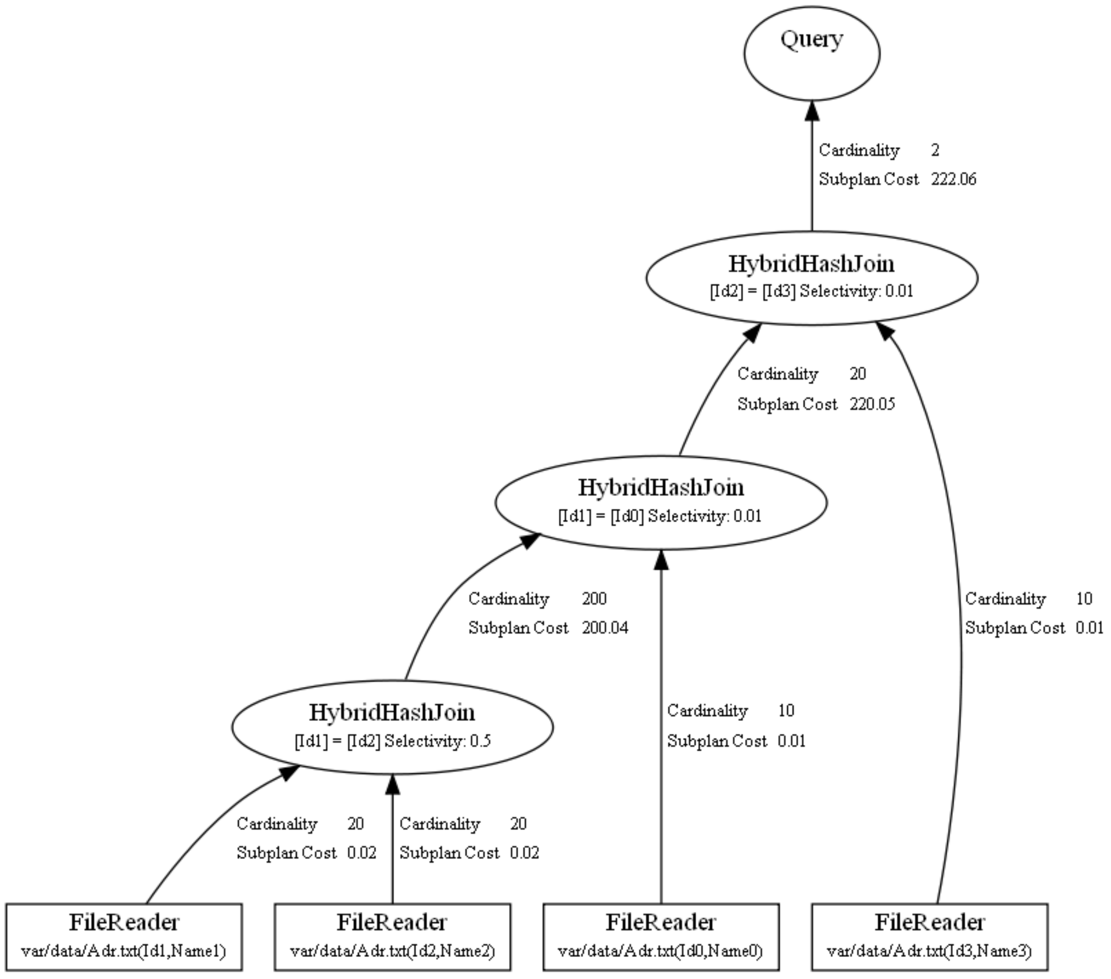


Figure 7.6: Query before join ordering optimization.

As input to the optimizer, we give it the worst case plan, $((\mathsf{R1} \bowtie \mathsf{R2}) \bowtie \mathsf{R0}) \bowtie \mathsf{R3}$, as shown in Figure 7.6. We can calculate the cost as follows, which we can see agrees with the figure.

$$\mathcal{C}(\mathsf{R1}) + \mathcal{C}(\mathsf{R2}) + \mathcal{C}(\mathsf{R0}) + \mathcal{C}(\mathsf{R3}) + \mathcal{C}(\mathsf{R1} \bowtie \mathsf{R2}) + \mathcal{C}(\mathsf{R1} \bowtie \mathsf{R0}) + \mathcal{C}(\mathsf{R2} \bowtie \mathsf{R3})$$
$$= 0.2 + 0.2 + 0.01 + 0.01 + 200 + 20 + 2 = 222.06$$

First, we tell the optimizer to only consider left-deep plans. It is not any point in doing so for this small query in real life, but for larger queries, we need to support left-deep exploration. The result can be seen in Figure 7.7. The optimizer has chosen to switch the order of the joins $\mathsf{R0} \bowtie \mathsf{R1}$ and $\mathsf{R1} \bowtie \mathsf{R2}$, as $\mathsf{R0} \bowtie \mathsf{R1}$ has much higher selectivity and thereby limits the cardinality of the temporary result in between them, keeping the cost down.

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 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.

Still, this plan has a much better cost of only 24.06 compared to 222.06. The query was optimized in 10.5 ms for the bushy plan and 2 ms for the left-deep plan.

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.

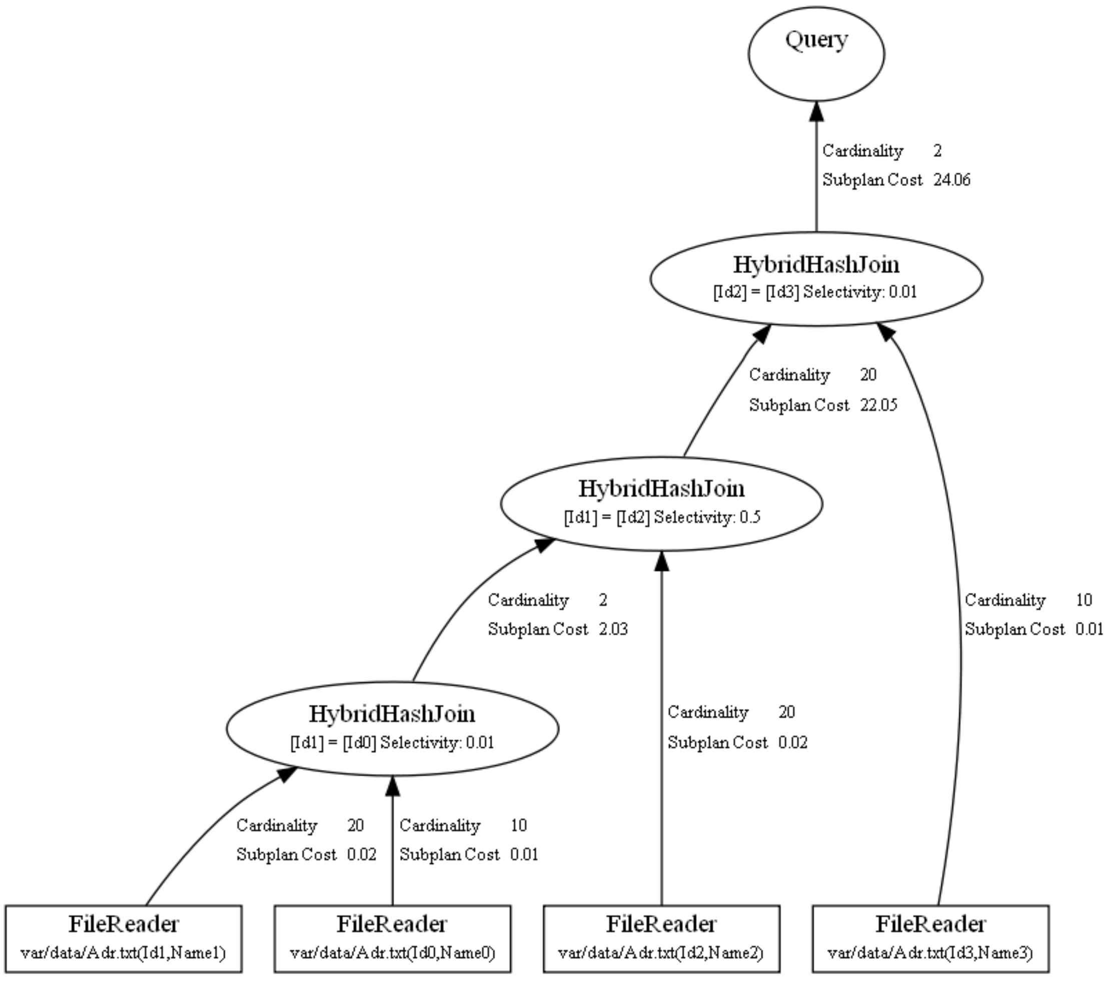


Figure 7.7: Query after optimization, only considering left-deep plans.

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, yielding a cost of only 6.06. See Figure 7.8

As we can see, the expected cardinality for the query is the same in both cases.

7.3.3 Multi-Query Optimization

As mentioned before, MARS supports multi-queries, which means that our optimizer should support optimizing such queries. So far, we have been focusing especially on this, but we have been careful to design and code to support it.

At this stage, the optimizer is able to reuse common subexpressions and create DAG-structured plans for multi-queries, but they are not optimal, merely a positive consequence of the design of the optimizer and physical plan generation routine. They are not optimal in the sense that the optimizer will not currently recognize common subexpressions and share equivalent subplans unless they are completely identical, and the cost model does not currently honor such plans as it should.

Still, we include a sample plan that may not be the optimal one, as an example of what is to come. See Figure 7.9.

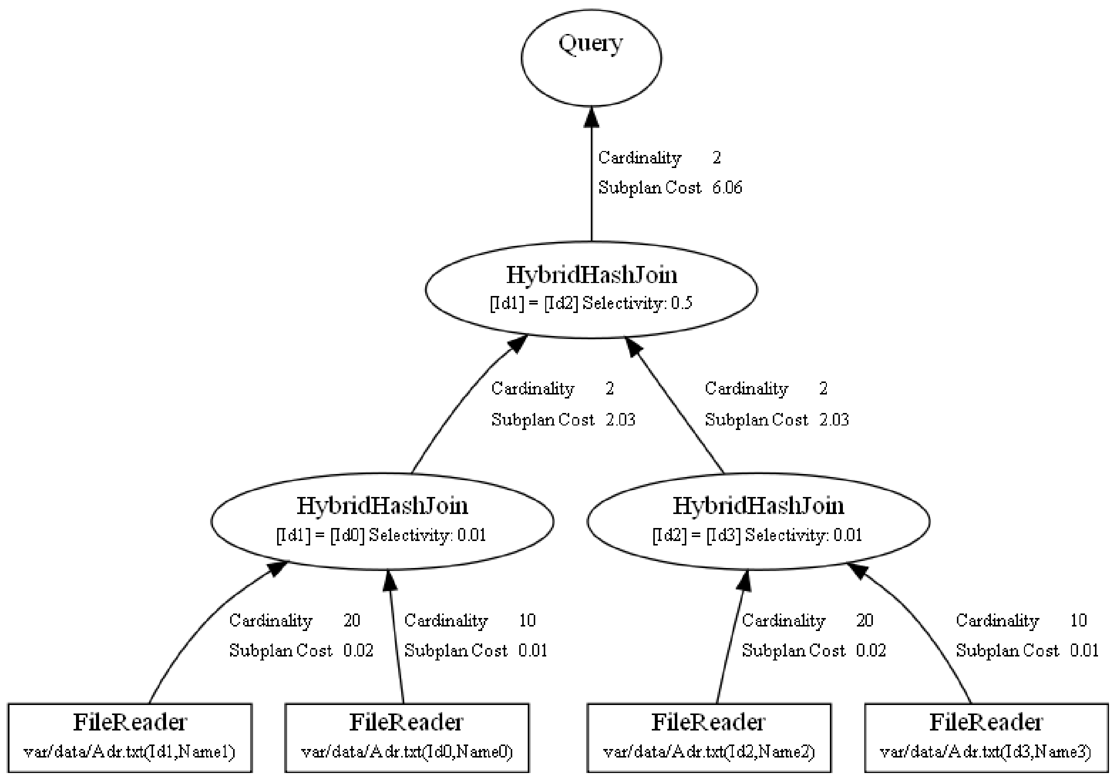


Figure 7.8: Query after optimization, also considering bushy plans.

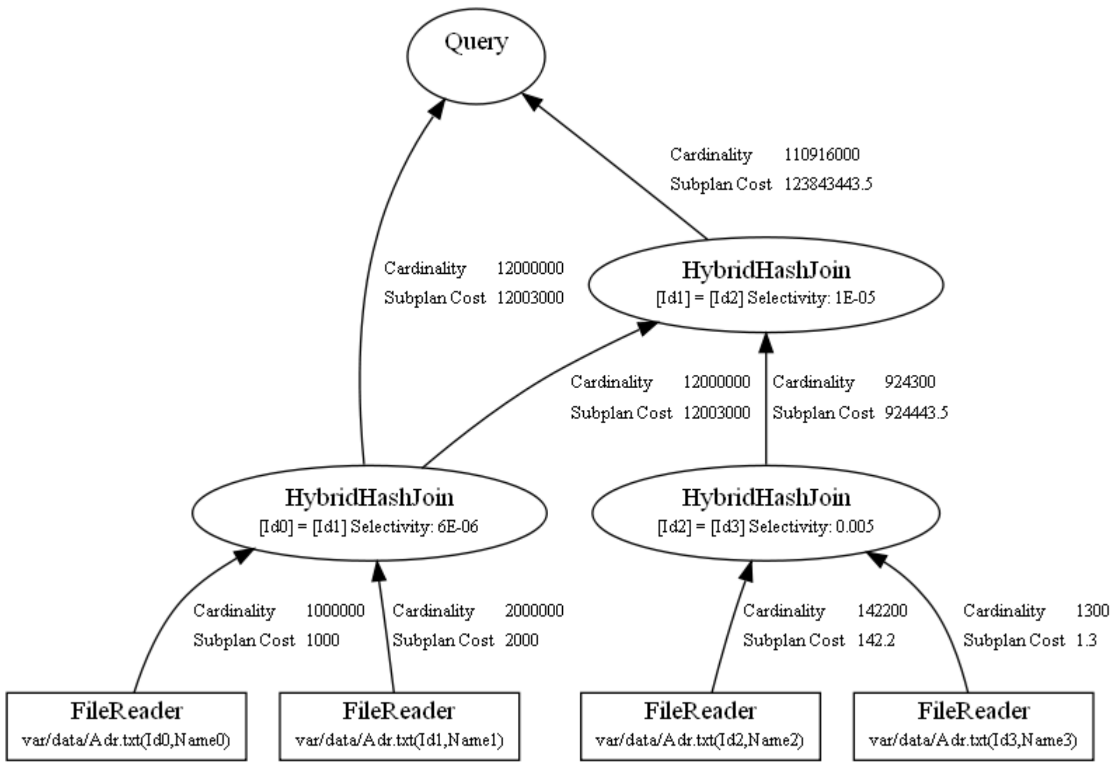


Figure 7.9: Sample multi-query utilizing common subexpressions.

7.3.4 Large Query with Selection

Figure 7.10 shows what larger queries might look like. This query is a chain join query with 7 relations, 6 joins and one selection (highlighted). We have included it mostly as a curiosity,

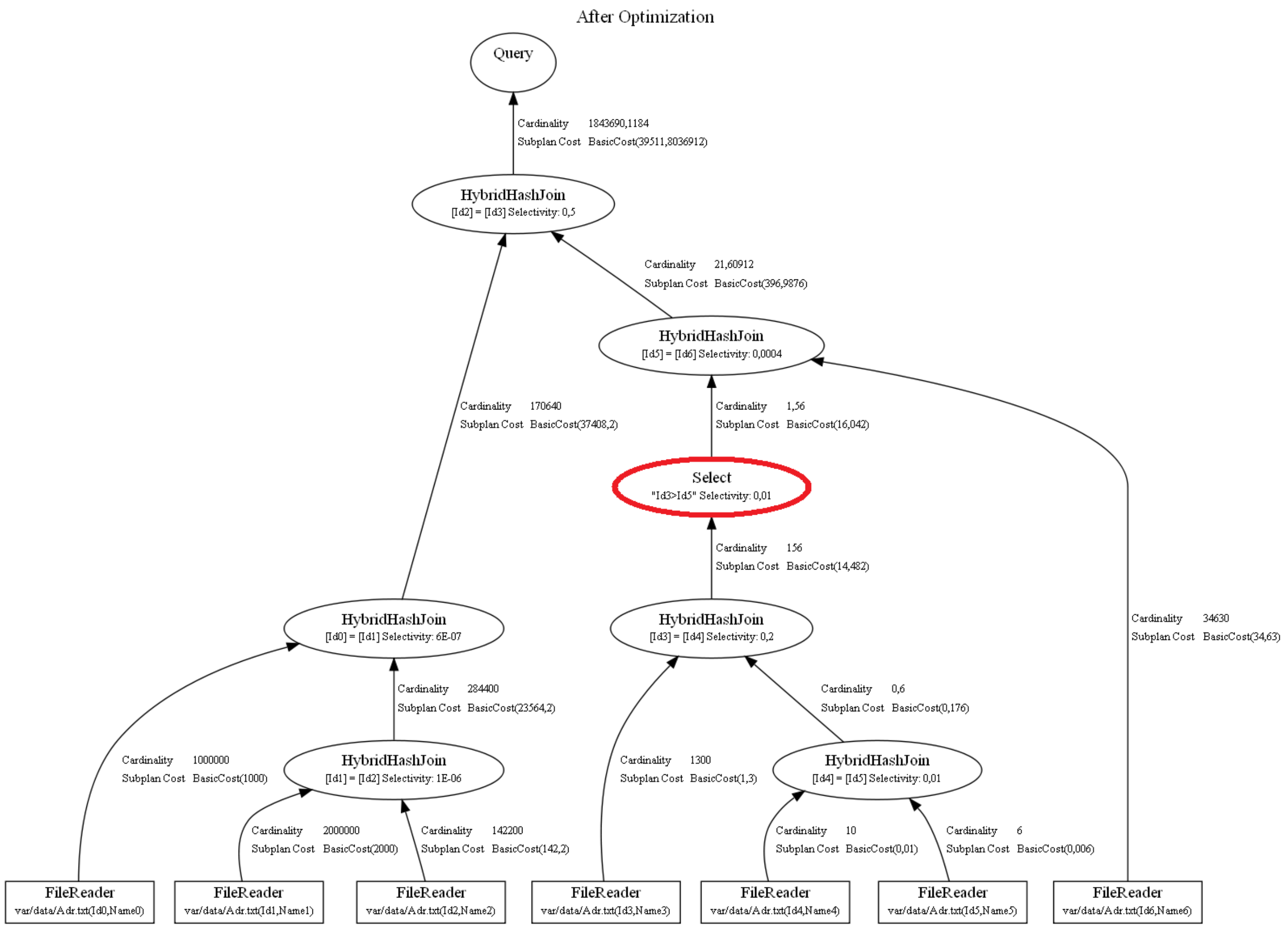

After Optimization
Query
Cardinality 1843690,1184
Subplan Cost BasicCost(39511,8036912)
HybridHashJoin
[Id2] = [Id3] Selectivity: 0,5
Cardinality 21,60912
Subplan Cost BasicCost(396,9876)
HybridHashJoin
[Id5] = [Id6] Selectivity: 0,0004
Cardinality 170640
Subplan Cost BasicCost(37408,2)
Cardinality 1,56
Subplan Cost BasicCost(16,042)
Select
"Id3>Id5" Selectivity: 0,01
Cardinality 156
Subplan Cost BasicCost(14,482)
HybridHashJoin
[Id0] = [Id1] Selectivity: 6E-07
HybridHashJoin
[Id3] = [Id4] Selectivity: 0,2
Cardinality 34630
Subplan Cost BasicCost(34,63)
Cardinality 284400
Subplan Cost BasicCost(23564,2)
Cardinality 0,6
Subplan Cost BasicCost(0,176)
Cardinality 1000000
Subplan Cost BasicCost(1000)
HybridHashJoin
[Id1] = [Id2] Selectivity: 1E-06
Cardinality 1300
Subplan Cost BasicCost(1,3)
HybridHashJoin
[Id4] = [Id5] Selectivity: 0,01
Cardinality 2000000
Subplan Cost BasicCost(2000)
Cardinality 142200
Subplan Cost BasicCost(142,2)
Cardinality 10
Subplan Cost BasicCost(0,01)
Cardinality 6
Subplan Cost BasicCost(0,006)
FileReader
var/data/Adr.txt(Id0,Name0)
FileReader
var/data/Adr.txt(Id1,Name1)
FileReader
var/data/Adr.txt(Id2,Name2)
FileReader
var/data/Adr.txt(Id3,Name3)
FileReader
var/data/Adr.txt(Id4,Name4)
FileReader
var/data/Adr.txt(Id5,Name5)
FileReader
var/data/Adr.txt(Id6,Name6)

Figure 7.10: Large bushy query after optimization, including selection (highlighted).

but it is worth noting that the optimizer found that a bushy plan was the best one, and that the selection was pushed as far down as it gets (due to the filter being Id3 > Id5).

The optimizer spent 24 ms optimizing this query, doing 654 rule appliances, generating 482 plans (roughly).

8 Conclusion and Further Work

8.1 Evaluation

In the introduction, we stated multiple goals. The first one was to *get a broad overview of ongoing efforts within the query optimization research field.* We have read quite a few papers and studied the source code of one leading open source database system and have gotten an overview over the different approaches to query optimization — especially rule based optimization.

The next goal was to *analyze the various approaches and techniques and justify their suitability for a future query optimizer for MARS.* We have concluded that rule based optimization is the way to go. We have decided to go for a combination of transformation based and construction based optimizer to get the best of both worlds; transformation for pre/post-processing, constructive for plan generation.

We also aimed to *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.* We believe we have found a design that satisfies the design goals. This is explained in Section 7.1. We have not laid out the design for everything, but explain why we can extend it to support things like orderings and DAGs. We also have a running optimizer based on the design, which shows its suitability and serves as a good foundation for further work. Even though much of the design of the plan generation step is based on [Neu05] and [NM08], we claim to have made a few enhancements, as mentioned in Section 5.5.

Finally, we wanted to *implement small parts of the architecture and implement some simple optimization rules. The implementation should lay the foundations for the work in the upcoming master thesis, and not be so simple it needs to be replaced.* We have a running optimizer than can do join ordering and selection pushdown. It also includes a simple transformation based rule. We also conducted performance tests that show acceptable performance, and we were able to visualize query plans, which is great for demonstration and debugging. On one hand, we have only implemented the most fundamental parts of the design, but they are still needed in the full implementation, so nothing will be thrown away. We still have a lot of work to do, but as can be seen in the following section about further work, we have the plans ready.

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.

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).

Integrable with MARS. Although the optimizer is not currently integrated with MARS, we have had access to an early version of MARS to avoid creating something that does not play well it. Both the optimizer and MARS are implemented in C# on .NET.

Extensibility. *fast* wanted an optimizer that is extensible, amongst other to support future operators. Our optimizer is extensible in terms of rules and cost model.

Future DAG-support. MARS supports DAGs, and so will our optimizer with a little bit of work.

Summarized, we would say we have reached the goals stated and are happy about the result. Not only do we have a running optimizer, but we have also documented the important parts of its inner workings. We are confident that our efforts during this project will pay off in the upcoming master thesis.

8.2 Further Work

In this section, we summarize further work for this project. Some of it, especially the first four sections, is work that needs to be done before the query optimizer actually starts becoming useful. The last four sections include possibilities it would be interesting to pursue, but is not critical.

For the upcoming semester, we hope to be able develop our current prototype into something that is integrated with MARS and is actually useful.

8.2.1 Integration with MARS

We have had access to a compiled version of MARS, but we have not currently integrated the optimizer with it. This is due to: 1) It was not a big gain at this stage — we need to get the optimizer design settled first. 2) It is smart to have as few dependencies as possible to isolate errors and easy debugging. 3) We have had some problems with our very early build MARS.

Still, it has been very useful to have it available to be able to create a design that is easy to integrate with MARS at a later stage. We plan to look at integration with MARS in the next semester. Things that need to be looked at in this context:

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. We need to introduce logical operators.

System catalogs. We will need to integrate the optimizer with the system catalogs in MARS to enable it to look up information on relations, indexes and create, store and look up statistics.

Plan caching. They query optimizer should cache generated query plans. However, we will need information from MARS to invalidate the cache, for instance when the schema changes. This can be quite tricky, for example due to different parameter selectivity.

Query handover. We already know how queries are represented in MARS' internal data structures, but we will need to determine how we actually pass the queries to and from MARS. Should we use MARS' (half-physical) parser or roll our own?

SparQL. Another pair of students are implementing a SparQL parser for MARS. It may be of interest to cooperate with them, as they can have solved the issue mentioned in the previous point.

8.2.2 Ordering and Grouping

In its current implementation, the optimizer does not take orderings and groupings into consideration. We will have to implement this to make the optimizer useful. [Neu05] describes a state machine to do this, which we plan to implement.

8.2.3 Operators

Support for more operators will need to implemented to enable the optimizer to handle the most common queries. Specifically, this includes sort, map, project and group by. We also need to implement predicate splitting as described in 7.2.2 to enable more advanced selection optimizations.

We also need to introduce logical operators (like Join) into MARS to enable true declarative queries.

8.2.4 Better Cost Model

The currently implemented cost model is very simple and does not take random vs sequential reads into account, let alone other dimensions like memory usage, CPU usage, network I/O cost, parallelization opportunities and so on. We will at least need to implement proper I/O modeling for it to be usable.

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.

8.2.5 Full Support for DAGs

We have limited support for DAGs today, but the architecture is extendable to support it. To do so, we need to implement detection of share equivalence and make the optimizer recognize common subexpressions. Some effort is also required to integrate this with MARS' execution model.

8.2.6 Optimizations

We have not focused too much on performance optimization except some profiling, so this is something we would look more into later. We have already mentioned heuristics and cost based pruning/incremental costing in Section 7.2.1 and transitive closures in Section 7.2.3. Further, we would try to generally optimize our code and look for logical optimizations (optimize the logic of the optimizer, not only code) as well.

8.2.7 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 7.3. 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.

Listing 8.1: SelectionRule.SearchBottomUp(), very simplified

```
public override void SearchBottomUp(PlanSet plans, BitSet current) {
    foreach (Plan inputPlan in plans) {
        Plan selectionPlan = new Plan(inputPlan) { Rule = this };
        plansCache[current | Produced].AddPlan( selectionPlan );
    }
}
```

8.2.8 Opportunities for Planner Parallelism

Currently, we cannot do much over 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 does no longer increase 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 relation 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, subplan requests can be queued to a thread pool. 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.

8.2.9 Parallel Execution and Costing

It is not only the optimization of query plans that can be parallelized — the execution of them can as well. Most high-end commercial DBMS-es produce parallel plans when it is beneficial. For example, sorting can be parallelized, or different parts of the query graph can be run of 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, as they will be executed more quickly. 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.

The current cost model interface cannot express this, so we will need to develop this further to enable the optimizer to consider parallel plans. [GHK92] mentions several techniques for query optimization of parallel execution. We may want to look into this in the upcoming semester.

A Code Samples

We have chosen not to include the full optimizer code base in the report, as it counts approximately 3500 lines of code. We refer to the accompanying CD for the full code base, which should be ready to compile and run. See Appendix B for a description of the CD-ROM contents.

The rest of this appendix include selected code samples that were to long to include in the main part of the report.

A.1 Rule Binder Initialization

As explained in 6.1, 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, as well as implements *IRuleBinder*. 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.

Listing A.1: Rule binder initialization.

```
private static void InitRuleBinders () {
    ruleBinderTypes = new List<Type>();
    foreach (Assembly assembly in AppDomain.CurrentDomain.GetAssemblies())
        foreach (Type type in assembly.GetTypes())
            if (Attribute.GetCustomAttribute(type, typeof( RuleBinderAttribute )) != null)
                ruleBinderTypes.Add(type);
}

private List<IRuleBinder> GetRuleBinders () {
    List<IRuleBinder> ruleBinders = new List<IRuleBinder>();
    foreach (Type type in ruleBinderTypes ) {
        IRuleBinder binderToAdd = (IRuleBinder) Activator. CreateInstance (type);
        ruleBinders.Add(binderToAdd);
    }
    return  ruleBinders ;
}
```

A.2 dot Language Sample

The following listing shows the GraphViz [AT 08] dot language used by the optimizer to visualize the operator graphs. This example is taken from the query in Section 7.3.1. For example, `28517321 [label="Query",shape=oval]` is the declaration of the topmost Query operator node, while `1547500 -> 28517321 [label="Cardinality: 12000000 Subplan Cost: 22003010"]` is the edge leading into it. The numbers are just unique identifiers for each node.

Listing A.2: dot file for query in Section 7.3.1.

```
digraph  test  {
  rankdir =BT
   labelloc =t
  label =" After␣Optimization "
  28517321 [ label ="Query",shape=oval ]
  1547500 [ label ="HybridHashJoin␣ Selectivity  :␣0.006" , shape=oval ]
  51067503 [ label =" Select ␣  Selectivity  :␣0.001" , shape=oval ]
  7506019 [ label =" FileReader ␣var / data /Adr. txt (Id1 ,Name1)",shape=box]
  7506019 −> 51067503  [ label =" Cardinality :␣1000000␣Subplan␣Cost:␣1000"]
  51067503 −> 1547500  [ label =" Cardinality :␣1000␣Subplan␣Cost:␣10001010"]
  46372056 [ label =" FileReader ,␣var / data /Adr. txt (Id2 ,Name2)",shape=box]
  46372056 −> 1547500  [ label =" Cardinality :␣2000000␣Subplan␣Cost:␣2000"]
  1547500 −> 28517321  [ label =" Cardinality :␣12000000␣Subplan␣Cost:␣22003010"]
  {  rank  = same;  7506019;46372056}
}
```

A.3 Selection Rule

To show how a rule is actually implemented in full, we have chosen to include the full implementation of the simplest search rule, *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*, 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. The code is an exact copy of the source code.

Listing A.3: SelectionRule implementation.

```
/// <summary>This rule  is  reponsible  for  constructing  selection  operators  in  query  plans .</
    summary>
public  class  SelectionRule  :  UnaryRule {
    public  SelectionRule (QueryOptimizer queryOptimizer)  :  base(queryOptimizer)
    { }

    /// <summary>Cost of  evaluating  the  predicates  for  each  record .</summary>
    public  double  PredicateCost  {  get;  set;  }
    /// <summary>User friendly  name for  this  rule .</summary>
    public  override  string  Name{ get  {  return  " Selection ";  }  }

    /// <summary>Updates the properties of the plan (costs, ordering, sharing).</summary>
    /// <param name="plan">The plan to update.</param>
    public override void UpdatePlan(Plan plan) {
        plan.Rule = this;
        // Todo: Sharing, ordering
        CalculateCosts(plan);
    }

    private void CalculateCosts(Plan plan) {
        BasicCost childCost = (BasicCost)plan.Children[0].Costs;
        BasicPlanSetState childState = (BasicPlanSetState)plan.Children[0].PlanSet.State;
        plan.Costs = childCost + new BasicCost(childState.Cardinality * PredicateCost);
    }

    /// <summary>Builds the Node in the physical algebra graph for this rule and calls
    /// its children recursively.</summary>
    /// <param name="plan">Plan to build algebra for.</param>
    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.Children[0].Rule.BuildAlgebra(plan.Children[0]);

        // Create node
        newNode = new Node() { OperatorType = Node.OperatorType };
        newNode.Children.Add(input);
        newNode.Properties = Node.Properties;
        newNode["SubplanCost"] = plan.Costs;
        newNode["Cardinality"] = ((BasicPlanSetState)plan.PlanSet.State).Cardinality;

        queryOptimizer.ReconstructionTable[plan] = newNode;
        return newNode;
    }
}
```

Listing A.4: Rule binder for SelectionRule.

```
/// <summary>Rule binder for SelectionRule.</summary>
[RuleBinder]
public class SelectionRuleBinder : AbstractRuleBinder<SelectionRule> {
    /// <summary>Node pattern to match in the operator graph during initialization.</summary
        >
    public override AbstractNodeMatcher Pattern {
        get { return new NodeTypeMatcher("SelectOperator"); }
    }

    /// <summary>Phase 2, initialize rules, set produced/required properties.</summary>
    public override void InitializeRules() {
        base.InitializeRules();
        foreach (SelectionRule rule in base.GetRules()) {
            if (rule.Node["Selectivity"] != null)
                rule.Selectivity = (double)rule.Node["Selectivity"];
            if (rule.Node["PredicateCost"] != null)
                rule.PredicateCost = (double)rule.Node["PredicateCost"];

            rule.Required = new List<BitSet>() { QueryOptimizer.BitSetManager.Empty };
            IRecordTypeDescriptor inputFieldTypes = rule.Node.InputTypeDescriptors[0];
```

```
                string expression = ((ExpressionProperty)rule.Node["Filter"]).Expression;
                // Todo: This can match substrings and is not reliable.
                // We're planning to use Fast's parser to do this.
                foreach (FieldOccurrence field in inputFieldTypes)
                    if (expression.Contains(field.Name))
                        rule.Required[0].AddAttribute(field.Name);
            }
        }
}
```

Listing A.5: UnaryRule implementation.

```
/// <summary>General base class for all unary rules (one input), offering
/// basic search functionality.</summary>
public abstract class UnaryRule : AbstractSearchRule {
    public UnaryRule(QueryOptimizer queryOptimizer) : base(queryOptimizer)
    { }

    /// <summary>Guides the search for this rule instance. Offers the basic search
        functionality
    /// for unary rules: generating all possible plans with this rule on the top.</summary>
    /// <param name="plans">PlanSet to add plans to (also defines goal properties).</param>
    /// <param name="limit">Abort the search if passing this cost limit (pruning).</param>
    public override void Search(PlanSet plans, ICost limit) {
        /* Get the possible input plans, i.e. plans with needed properties,
         * except the ones we produce ourselves.
         */
#if DIAGNOSTICS
        Debug.Print("UnaryRule␣producing␣{0}␣searching␣for␣{1}.", Produced, plans.Properties
            - Produced);
#endif
        PlanSet input = queryOptimizer.GeneratePlans(plans.Properties - Produced, limit);
        if (input == null)
            return; // No plans

        if (plans.Count == 0) {
            // First plan, so set some state.
            plans.State = new BasicPlanSetState() {
                Cardinality = ((BasicPlanSetState)input.State).Cardinality * Selectivity,
                TupleSize = ((BasicPlanSetState)input.State).TupleSize
            };
        }

        // Add each input plan to the PlanSet, updating their properties.
        foreach (Plan plan in input) {
            Plan newPlan = new Plan();
            newPlan.Children = new List<Plan> { plan };
            UpdatePlan(newPlan);
            plans.AddPlan(newPlan);
        }
    }
}
```

Listing A.6: AbstractSearchRule implementation.

```
/// <summary>Base class for search rules, implementing required members and Filter caching
    .</summary>
public abstract class AbstractSearchRule : ISearchRule {
    private BitSet? cachedFilter;
    protected QueryOptimizer queryOptimizer;

    public AbstractSearchRule (QueryOptimizer queryOptimizer) {
        this.queryOptimizer = queryOptimizer;
        this.Id = queryOptimizer.GetRuleNumber(this);
        this. Selectivity = 1;
    }

    /// <summary>Node that this rule was instantiated from.</summary>
    public Node Node{ get; set; }
    /// <summary> Selectivity for this rule. 1 for non-limiting rules.</summary>
    public double Selectivity { get; set; }

    #region ISearchRule Members

    /// <summary>Properties produced by this rule.</summary>
    public BitSet Produced { get; set; }
    /// <summary>List of properties required for this rule.</summary>
    public IList<BitSet> Required { get; set; }

    /// <summary>Automatically generates and caches the Filter property based
    /// on the Produced and Required properties.</summary>
    public virtual BitSet Filter {
        get {
            if (cachedFilter == null) {
                cachedFilter = Produced;
                foreach (BitSet bitset in Required)
                    cachedFilter |= bitset;
            }
            return cachedFilter.Value;
        }
    }

    /// <summary>Determine if this rule is relevant to reach the given goal.</summary>
    public bool IsRelevantTo (BitSet goal) {
        return Filter <= goal;
    }

    /// <summary>Guides the search for this rule instance.</summary>
    /// <param name="plans">PlanSet to add plans to (also defines goal properties).</param>
    /// <param name="limit">Abort the search if passing this cost limit (pruning).</param>
    public abstract void Search(PlanSet planSet, ICost limit);

    #endregion

    #region IRule Members

    /// <summary>Id for this rule as given by the optimizer.</summary>
    public int Id { get; set; }
    /// <summary>User friendly name for this rule.</summary>
    public abstract string Name{ get; }
    /// <summary>Updates the properties of the plan (costs, ordering, sharing).</summary>
    /// <param name="plan">The plan to update.</param>
    public abstract void UpdatePlan(Plan plan);
    /// <summary>Builds the Node in the physical algebra graph for this rule and calls
    /// its children recursively.</summary>
    /// <param name="plan">Plan to build algebra for.</param>
    public abstract Node BuildAlgebra(Plan plan);

    #endregion
```

```
}
```

Listing A.7: AbstractRuleBinder implementation.

```
/// <summary>Base class for rule binders, offering common functionality.</summary>
/// <typeparam name="RuleType">The type of rule to bind.</typeparam>
public abstract class AbstractRuleBinder<RuleType> : IRuleBinder where RuleType :
    IProducerRule {
    protected List<RuleType> rules = new List<RuleType>();
    /// <summary>Node pattern to match in the operator graph during initialization.</summary
        >
    public abstract AbstractNodeMatcher Pattern { get; }
    /// <summary>Query optimizer initializing the rules, set by the optimizer itself.</
        summary>
    public QueryOptimizer QueryOptimizer { get; set; }

    /// <summary>Phase 1, initialize bit property sets.</summary>
    /// <param name="matches">Collection of matching nodes in the operator graph.</param>
    public virtual void InitializeBitSets (IEnumerable<PatternMatch> matches) {
        // For each match, instantiate a rule and set properties.
        foreach (PatternMatch match in matches) {
            RuleType rule = CreateRule(match.Sources[0]);
            QueryOptimizer.BitSetManager.AddProduced(
                new HashSet<string>() { BitSetManager.RuleApplied(rule.Id) });
            rules.Add(rule);
        }
    }

    private RuleType CreateRule(Node node) {
        RuleType rule = (RuleType) Activator.CreateInstance (typeof(RuleType), QueryOptimizer)
            ;
        rule.Node = node;
        node.Rules = new List<IProducerRule>() { rule };
        return rule;
    }

    /// <summary>Phase 2, initialize rules, set produced/required properties.</summary>
    public virtual void InitializeRules () {
        foreach (RuleType rule in rules)
            rule.Produced = QueryOptimizer.BitSetManager.GetWithValues(
                BitSetManager.RuleApplied(rule.Id));
    }

    /// <summary>Phase 3, returns the initialized rules to the optimizer.</summary>
    public virtual IEnumerable<IProducerRule> GetRules() {
        return rules.Cast<IProducerRule>();
    }
}
```

A.4 MergeTrimSort Rule

This is the full implementation of the MergeTrimSort rule. See Section 6.2.2 for an explanation.

Listing A.8: MergeTrimSort rule implementation.

```
[TransformationRule(TransformationType.Pre)]
public class MergeTrimSort : AbstractPreprocessor {
    public override AbstractNodeMatcher Pattern {
        get {
            return (new NodeTypeMatcher("TrimOperator"))
                        .GroupAs("trim")
                        .WithChildren(
                            new ZeroOrMore(
                                NodeBehaviourMatcher.All(OperatorBehaviour.SetPreserving |
                                    OperatorBehaviour.OrderPreserving)
                                .WithAnyOneParent() // Don't match a branching node.
                            )
                            .WithChildren(
                                (new NodeTypeMatcher("SortOperator")).GroupAs("sort")
                            )
                        );
        }
    }

    public override void Fire(PatternMatch match) {
        Node trimOperator = match.Groups["trim"].OnlyMatch;
        Node sortOperator = match.Groups["sort"].OnlyMatch;

        match.Groups["trim"].OnlyMatch = null;

        int trimOffset  = (int)trimOperator["offset"];
        int sortOffset  = (int)sortOperator["offset"];
        int finalOffset = trimOffset + sortOffset;

        int trimHitCount = (int)trimOperator["hitcount"];
        int sortHitCount = (int)sortOperator["hitcount"];
        int finalHitCount = -1;

        Debug.Assert(sortHitCount != 0 || trimHitCount != 0,
                "This␣query␣will␣not␣return␣any␣results␣and␣should␣be␣replaced␣with␣a␣NOOP.");

        if (trimHitCount > 0)
            finalHitCount = trimHitCount;

        if (sortHitCount > 0)
            finalHitCount = Math.Min(finalHitCount, Math.Max(sortHitCount - trimOffset, 0));

        sortOperator["offset"] = finalOffset;
        sortOperator["hitcount"] = finalHitCount;
    }

    public override bool Iterative { get { return true; } }
}
```

A.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.

A.5.1 BitSet Implementation

Listing A.9 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.

Listing A.9: BitSet private data.

```
/// <summary>The bits are internally stored as ints. 1 int up to 32, 2 up to 64 etc.</
    summary>
private int[] data;
/// <summary>Current number of bits stored.</summary>
private int length;
```

Listing A.10 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.

Listing A.10: BitSet single property interface.

```
public void Add(string property) {
    this[manager[property]] = true;
}
public void Remove(string property) {
    this[manager[property]] = false;
}
public bool Contains(string property) {
    return this[manager[property]];
}
```

Listing A.11 shows how the [] operator used in the previous listing is implemented. First, the correct array index is found by computing `index / 32` (remember, 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 of unioned *into* the stored bit mask.

Listing A.11: BitSet internal property implementation.

```
private bool this[int index] {
    get {
        // Get the correct array item, AND with 1 bitshifted
        // to the correct position and return if it is not 0.
        return ((this.data[index / 32] & (((int)1) << (index % 32))) != 0);
    }
    set {
        if (value)
            // OR with 1 bitshifted to the correct position.
            this.data[index / 32] |= ((int)1) << (index % 32);
        else
            // AND with NOT (1 bitshifted to the correct position).
            this.data[index / 32] &= ~(((int)1) << (index % 32));
    }
}
```

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.

Listing A.12: BitSet set intersection operator.

```
public static BitSet operator &(BitSet a, BitSet b)
{
    // Intersect all the data items (&) and construct a new BitSet.
    int length = (a.length + 31) / 32;
    int[] data = new int[length];
    for (int i = 0; i < length; i++)
        data[i] = a.data[i] & b.data[i];

    return new BitSet(a.manager, data, a.length);
}
```

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.

Listing A.13: BitSet IsSubSet operator.

```
public static bool operator <=(BitSet a, BitSet b) {
    // Subset (A <= B) is implemented as (A & !B) == EMPTY.
    int length = (a.length + 31) / 32;
    for (int i = 0; i < length; i++)
        if ((a.data[i] & ~b.data[i]) > 0)
            return false;

    return true;
}
```

A.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 $\{\{a_1, a_2\}, b_1, c_1\}$.

Later we plan to extend it to also prune properties that are never required or never produced.

Listing A.14: BitSet Minimization

```
/// <summary>Prepares the BitSetManager for use. This must be called before any bitsets can
    be used.
/// It will minimize the bit set properties and register all the mappings.</summary>
public void Prepare() {
     ValidatePrepared(false);
    prepared = true;
    Dictionary<string, string> mappings = MinimizeProperties();
    RegisterMappings(mappings);
}

/// <summary>Minimize the properties. Currently, this includes merging all properties
/// that are always produced together.</summary>
/// <returns>A dictionary that maps property name -> property name.
/// If for instance A and B are always produced together, it will contain {A->A, B->A}.</
    returns>
 private Dictionary<string, string> MinimizeProperties() {
    Dictionary<string, string> mappings = new Dictionary<string, string>();
    // Register all direct mappings in all produced sets by default.
    foreach (NestableHashSet<string> producedSet in produced)
```

```
        foreach ( string property in producedSet)
            mappings[ property ] = property ;

    HashSet< string > allProducedProperties = new HashSet< string >() ;
    // Get all the produced properties unioned together .
    foreach ( NestableHashSet < string > producedSet in produced)
        allProducedProperties .UnionWith(producedSet);

    // Now, foreach over all properties , visiting each combination {propA, propB} once,
    // where propA < propB.
    foreach ( string propA in allProducedProperties ) {
        foreach ( string propB in allProducedProperties ) {
            if (propA.CompareTo(propB) < 0) {
                // Foreach over all produced sets . If a set contains propA or propB, add it
                // to our set of sets .
                HashSet<NestableHashSet< string >> setsContainingA = new HashSet<
                    NestableHashSet< string >>();
                HashSet<NestableHashSet< string >> setsContainingB = new HashSet<
                    NestableHashSet< string >>();
                foreach ( NestableHashSet < string > p in produced) {
                    if (p. Contains (propA))
                        setsContainingA .Add(p);
                    if (p. Contains (propB))
                        setsContainingB .Add(p);
                }
                // Now setsContainingA contains all sets containing A and setsContainingB
                // contains all sets containing B.
                // If these two sets are set equals (contain the same elements ), it means
                // that propA and propB are always produced together .
                if ( setsContainingA . SetEquals ( setsContainingB )) {
                    // If so, merge the properties by making propB map to whatever propA
                    // maps to. The reason we're mapping propB to mappings[propA] and
                    // not to propA, is that propA could have been remapped earlier itself .
                    mappings[propB] = mappings[propA];
                }
            }
        }
    }
    return mappings;
}

/// <summary>Registers the minimized mappings. This means adding entries
/// to the nameToIndex and indexToName dictionaries .</summary>
/// <param name="mappings">The mappings to register .</param>
private void RegisterMappings ( Dictionary < string , string > mappings) {
    // mappings. Values contains all possible bit property variations .
    // Register a physical property for all these .
    foreach ( string property in mappings. Values ) {
        // Only add each possible variation once.
        if (!nameToIndex.ContainsKey( property )) {
            nameToIndex[ property ] = indexToName.Count;
            indexToName.Add(new HashSet<string>() { property });
        }
    }

    // Then, add all the remappings (i.e. {B−>A}).
    foreach ( KeyValuePair < string , string > pair in mappings) {
        // Get the target index
        int index = nameToIndex[ pair . Value ];
```

```
            // Make the from-property point to the correct index
            nameToIndex[pair.Key] = index;
            // Add the from-property to the reverse lookup table
            indexToName[index].Add(pair.Key);
        }
}
```

A.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 following test constructs the query on page 38 (PDF page 55) in [Moe06] and is the one used to generate the figures in Section 7.3.2. It feeds the optimizer the worst possible plan (cost 222), and asks the optimizer to optimize it using bushy enumeration. Afterwards it verifies that the resulting plan has the correct cost (6.06) 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 (24.06) and is correctly laid out.

Listing A.15: SimpleJoins automated test.

```
[Test]
public void SimpleJoins()
{
    double[] cardinalities = new double[] { 10, 20, 20, 10 };
    double[] selectivities = new double[] { 0.5, 0.01, 0.01 };

    // We input the worst plan possible with cost 222.
    TestNode[] scans = new TestNode[] { GetFileReaderScan(cardinalities[0], 0),
                                        GetFileReaderScan(cardinalities[1], 1),
                                        GetFileReaderScan(cardinalities[2], 2),
                                        GetFileReaderScan(cardinalities[3], 3) };
    TestNode[] joins = new TestNode[3];
    joins[0] = GetJoin("Id1", "Id2", selectivities[0], scans[1], scans[2]);
    joins[1] = GetJoin("Id1", "Id0", selectivities[1], joins[0], scans[0]);
    joins[2] = GetJoin("Id2", "Id3", selectivities[2], joins[1], scans[3]);
    TestNode query = GetQuery(joins[2]);

    // Now test bushy enumeration. This should yield the plan (0 X 1) X (2 X 3)
    // with cost 6.06
    TestNode[] expectedScans = new TestNode[] { GetFileReaderScan(cardinalities[0], 0, true)
        ,
                                        GetFileReaderScan(cardinalities[1], 1, true),
                                        GetFileReaderScan(cardinalities[2], 2, true),
                                        GetFileReaderScan(cardinalities[3], 3, true) };
    TestNode[] expectedJoins = new TestNode[3];
    expectedJoins[0] = GetJoin("Id1", "Id0", selectivities[0], expectedScans[0],
        expectedScans[1], true);
    expectedJoins[1] = GetJoin("Id2", "Id3", selectivities[2], expectedScans[2],
        expectedScans[3], true);
    expectedJoins[2] = GetJoin("Id1", "Id2", selectivities[1], expectedJoins[0],
        expectedJoins[1], true);
    TestNode expected = GetQuery(expectedJoins[2]);
    expectedJoins[2]["SubplanCost"] = new BasicCost(6.06);

    queryOptimizer.JoinEnumeration = JoinEnumeration.Bushy;
    Node result = queryOptimizer.Optimize(query);
    expected. RecursiveAssert ( result );

    // Now test left-deep enumeration. This should yield the plan ((0 X 1) X 2) X 3
    // with cost 24.06
    expectedJoins [0] = GetJoin("Id1", "Id0", selectivities [0], expectedScans [0],
        expectedScans [1], true);
    expectedJoins [1] = GetJoin("Id1", "Id2", selectivities [1], expectedJoins [0],
        expectedScans [2], true);
    expectedJoins [2] = GetJoin("Id2", "Id3", selectivities [2], expectedJoins [1],
        expectedScans [3], true);
    expected = GetQuery( expectedJoins [2]) ;
    expectedJoins [2][ "SubplanCost"] = new BasicCost (24.06) ;

    queryOptimizer.JoinEnumeration = JoinEnumeration.LeftDeep;
    result = queryOptimizer.Optimize(query);
    expected. RecursiveAssert ( result );
}
```

B CD-ROM

The accompanying CD-ROM 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.

B.1 How to Run the Optimizer

B.1.1 Screencast

A short screencast demonstrating the optimizer can be found at
`http://www.screencast.com/t/kUI6SbbgLM`.

B.1.2 Running the Binaries

We are unable to provide running binaries of the optimizer, as they currently depend on assemblies from *fast*, which we were not allowed to distribute.
To be able to test the optimizer, please contact Øystein Torbjørnsen (`Oystein.Torbjornsen@microsoft.com`)

To be able to run the optimizer, you will need Microsoft Windows with .NET Framework 3.5 installed.

The optimizer is currently not integrated with MARS and only offers a very simple console interface to run five pre-defined test cases:

Simple Query Runs optimization of a simple query with one join and one selection.

Selections Runs optimization of a query with n (choose) selections in sequence.

Simple Joins Runs optimization of a query with two joins, the same query as used in section 7.3.2, both bushy and left-deep.

Chain Joins Runs optimization of a chain join query with n (choose) relations, both bushy and left-deep.

Speed Test Times the result of 10 runs of 1..15 joins for both chain and star joins, bushy and left-deep. Ends with a timings summary with averages in milliseconds for all test cases.

This is how to make it run:

1. Copy the full contents of
   Binaries to a folder on your hard drive. This is important as it will be writing files to the current directory.

2. Run OptimizerProfiling.exe. You will be presented with a few choices.

3. After you give it some input, it should print some text to the console, blink two console windows and open two images; the query before and after optimization. The images are left on disk in the current directory.

B.1.3 Building and Running

The supplied source code will not build since it depends on assemblies from *fast*, which we were not allowed to distribute.

To be able open, build and run the optimizer, you will need Microsoft Windows with Visual Studio 2008 installed. The source code on the CD is self-contained and includes all external dependencies, like NUnit.

This is how to compile and run it:

1. Copy the full contents of
   Source to your hard drive.

2. Open the solution **Optimizer.sln** in Visual Studio.

3. Make sure OptimizerProfiling is set as the start up project.

4. Run the solution by hitting F5 or clicking run in Visual Studio.

Feel free to explore the source code and set breakpoints anywhere to figure out its inner workings.

Bibliography

[AT 08] AT and T Research. Graphviz - graph visualization software. `http://www.graphviz.org/`, 2008.

[Bil] Keith Billings. A tpc-d model for database query optimization in cascades. `http://web.cecs.pdx.edu/~kgb/t/title.shtml`.

[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.

[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.

[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.

[Neu05] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs.* PhD thesis, Mannheim, 2005.

[NHM05] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 490–501, Washington, DC, USA, 2005. IEEE Computer Society.

[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.

[Sto87] Michael Stonebraker. The design of the postgres storage system. pages 289–300, 1987.

[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.