Efficient Generation and Execution of DAG-Structured Query Graphs

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Wirtschaftsinformatiker
Thomas Neumann
aus Köln

Mannheim 2005

Dekan:	Professor Dr. Matthias Krause, Universität Mannheim
Referent:	Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent:	Professor Dr. Bernhard Seeger, Universität Marburg

Tag der mündlichen Prüfung: 1. Juli 2005

Abstract

Traditional database management systems use tree-structured query evaluation plans. While easy to implement, a tree-structured query evaluation plan is not expressive enough for some optimizations like factoring common algebraic subexpressions or magic sets. These require directed acyclic graphs (DAGs), i.e. shared subplans.

This work covers the different aspects of DAG-structured query graphs. First, it introduces a novel framework to reason about sharing of subplans and thus DAG-structured query evaluation plans. Second, it describes the first plan generator capable of generating optimal DAG-structured query evaluation plans. Third, an efficient framework for reasoning about orderings and groupings used by the plan generator is presented. And fourth, a runtime system capable of executing DAG-structured query evaluation plans with minimal overhead is discussed.

The experimental results show that with no or only a modest increase of plan generation time, a major reduction of query execution time can be achieved for common queries. This shows that DAG-structured query evaluation plans are serviceable and should be preferred over tree-structured query plans.

Zusammenfassung

Traditionelle Datenbankmanagementsysteme verwenden baumstrukturierte Ausführungspläne. Diese sind effizient und einfach zu implementieren, allerdings nicht ausdrucksstark genug für einige Optimierungstechniken wie z.B. die Faktorisierung von gemeinsamen algebraischen Teilausdrücken oder magic sets. Diese Techniken erfordern gerichtete azyklische Graphen (DAGs), d.h. gemeinsam verwendete Teilpläne.

Die Arbeit behandelt die verschiedenen Aspekte von DAG-strukturierten Anfragegraphen. Zunächst wird ein formalen Modell zum Schließen über gemeinsam verwende Teilpläne und damit über DAG-strukturierte Anfragepläne vorgestellt. Anschließend wird dieses Modell in einem Plangenerator zur Erzeugung von optimalen DAG-strukturierten Anfrageplänen verwendet; bisherige Ansätze konnten die optimale Lösung nicht garantieren. Weiterhin wird eine neue Datenstruktur beschrieben, die dem Plangenerator eine effiziente Verwaltung von Sortierungen und Gruppierungen ermöglicht. Schließlich wird ein Laufzeitsystem vorgestellt, das die Ausführung von DAG-strukturierten Anfrageplänen mit sehr geringem Mehraufwand relativ zu baumstrukturierten Anfrageplänen ermöglicht.

Die experimentellen Ergebnisse zeigen, dass ohne bzw. mit nur etwas höherem Zeitaufwand im Plangenerator DAG-strukturierte Anfragepläne erzeugt werden können, die für übliche Anfragen eine erheblich Reduzierung der Ausführungszeit bewirken können. Diese Ergebnisse zeigen, dass DAG-strukturierte Anfragepläne mit vertretbarem Aufwand allgemein einsetzbar sind und deshalb anstelle von baumstrukturierten Anfrageplänen verwendet werden sollten.

Contents

1. Introduction

Queries are usually the most important way to access the data contained in a database. The user provides a declarative description of the data he needs, and the database management system retrieves the data specified by the query. This query processing is roughly done in a two-step approach: First, the system determines the best way to execute the declarative query (converting it into a more imperative form) and then executes it, producing the requested data.

Traditionally, the output of the first step is an operator tree [50]. Each of the operators takes the output of its children as input, performs some operation on it, and produces a new output stream that contains the intermediate result. Finally, the root of the operator tree produces the result of the whole query. These trees have many nice properties and are easily handled but they also have a strong limitation: the output of an operator can only be used by a single other operator. As a consequence, intermediate results cannot be reused by multiple operators, which is very unfortunate.

One possible solution is to use directed acyclic graphs (DAGs) instead of trees. When building query graphs as DAGs, operators can easily reuse intermediate results, as they can share children. In fact, this structure is common for many problems: examples include multi-query optimizers [41], data migration processes [2] that factorize and share common subqueries, optimization of disjunctive queries with bypass plans [73], or parallel query processing [13]. Query processing over data streams also relies upon DAG-structured query execution plans [42].

However, using DAGs instead of trees is much more complex. Therefore, existing query optimizers usually limit themselves to trees [44] or only consider very limited forms of DAGs [20]. In this work, we present a novel approach that allows the use of arbitrary DAGs during query processing. We concentrate on the different aspects of query optimization in the presence of DAGs, but also consider the problem of executing DAGs. The work intends to describe every part of the database management system that has to be changed in order to support DAGs. For the experimental results, a prototype system was actually implemented.

The rest of the thesis is structured as follows: First, query processing in general is described in Chapter 2. Chapter 3 and 4 discuss related work. Chapter 5 and 6 describe the core of the query optimizer, while Chapters 8, 7 and 9 describe different components of the query optimizer. The actual execution of DAGs is described in Chapter 10. Chapter 11 presents experimental results. Finally, conclusions and future work are discussed in Chapter 12.

2. Query Processing

2.1. Overview

Most database management systems (DBMS) offer a query interface, which allows the user to retrieve data by specifying a declarative query (e.g. in SQL, OQL etc.). Declarative means that the user specifies in which data he is interested but not how this data should be retrieved or computed. This enables the DBMS to choose among multiple possible ways to answer the query, which can have significantly different runtime characteristics. As a consequence, query processing in a DBMS is usually structured as shown in Figure 2.1. The query given is first processed by the compile time system, which analyzes the query and tries to find the best way to answer it. The output of this step is an execution plan that is passed to the runtime system, which actually executes the plan and produces the query result. This work concentrates on the compile time system, but also touches some aspects of the runtime system. This chapter provides an overview of these two components.

2.2. Compile Time System

The main task of the compile time system is to convert the query into the "best" execution plan that produces the requested data. The exact notion of "best" depends on the application and is discussed in Chapter 9. This process is quite involved and, therefore, split into several processing steps that are discussed next.

The basic steps of query compilation are shown in Figure 2.2. While the details vary among different query compilers, as some omit or combine steps, query compilation consists roughly of the following steps: First, the query string provided by the user is *parsed* and converted into an abstract syntax tree. This

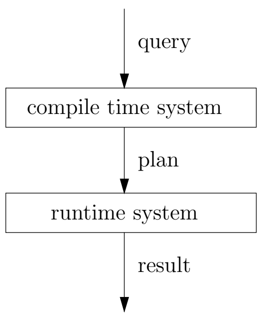

Figure 2.1.: Phases of query processing

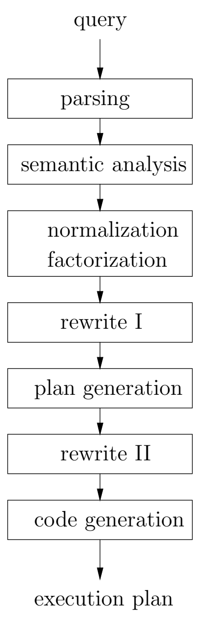

Figure 2.2.: Overview of query compilation

abstract syntax tree is examined during the *semantic analysis* and converted into a logical representation of the query. After this step, the query compiler has detected any errors in the query and has generated a concise logical representation of the semantics of the query. The exact representation varies from system to system, some possibilities are relational calculus [11], tableaus [1], monoids [14], algebras [24], or combinations thereof. What these representations have in common is that they provide a precise formal description of the data requested by the user, still without determining how to compute the data. The following steps are mostly optional, as they do not influence the semantics of the query but only the efficiency of its execution. Besides, the following steps are (or at least could be) query language independent, as they only care about the logical representation of the query and not the query itself.

In the following, we assume that this logical representation is an expression in the logical algebra, which will be transformed into an expression in the physical algebra by the plan generator. Some query compilers (e.g. [24]) already mix logical and physical algebra before plan generation, but this only has a minor influence on query compilation in general.

After constructing the logical representation, the query is first *normalized* and afterwards common simple subexpressions are *factorized* (e.g., if the expression "$5*x$" occurs twice, the result is reused). This step can include constant folding, construction of conjunctive normal forms etc. The goal is to remove as many computations from the query as possible, either by performing the computation at compile time or by finding shared expressions that only have to be computed once.

The next step is called *rewrite I* and consists of transformation rules that are applied to the query. These transformations can have a large impact on the quality of the resulting plan but are too difficult to model in the plan generation step. They are done beforehand and are usually not cost-based. The most important transformations are query unnesting [57], view resolution and view merging [45] and predicate push-down or pull-up [56].

The *plan generation* step is the core of the optimization phase. The details of this step are discussed in the next section. Basically, the plan generator takes the logical representation of the query and transforms it into a physical plan. While the logical plan describes the semantics of the query, the physical plan describes the steps required to compute the result of the query. This transformation is cost-based, i.e., the plan generator tries to construct the plan that will produce the query result with minimal costs. Costs are used to estimate the runtime behavior of a physical plan. See Section 9.3.2 for a discussion of different cost terms.

This physical plan is transformed again in the *rewrite II* phase. This is the analogue to the rewrite I phase operating on physical plans. Theoretically, both rewrite I and rewrite II are redundant, as the plan generator could consider the different alternatives and choose the cheapest one. However, for practical purposes it is often difficult to describe the transformations in a form suitable for the plan generator. Besides, these transforms could have a large impact on the search space. Therefore, these transformation steps are used as heuristics to improve the constructed plan. Typical transformations done in the rewrite II phase are group-by push-down [80], predicate pull-up [33, 46] and merging of successive selections and maps.

Finally, the *code generation* step transforms the physical plan into a form that can be executed by the runtime system of the database management system. This step might do nearly nothing if the database is able to execute the physical plan directly. However, database systems often require additional code to test predicates etc. This can be machine code [51], code for a virtual machine [78], interpreted expression trees [31] or some other form suitable for execution [15].

After this step the query compilation is done, as the query has been transformed into a form that can be executed by the database management system.

2.3. Runtime System

The runtime system manages the actual database instance and can execute the plans generated by the compile time system. The actual implementation differs between systems, type relevant parts of a typical runtime system architecture are shown in Figure 2.3: The lowest layer consists of the *physical storage* layer. It interacts with the storage media and organizes the storage space into *partitions*. Based upon that, the *logical storage* layer maintains *segments*. Like files in a file system, segments offer growable storage space that abstracts from physical storage, fragmentation etc. The segments themselves are organized into *pages*, which are read and written on demand by the *buffer manager*. These pages have a fixed size, which makes I/O and buffer management simple. As the pages are typically much larger than the data elements stored on it and the data can vary in size, pages offer a simple storage space management that

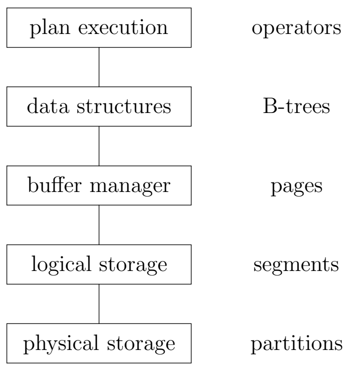

Figure 2.3.: Overview of query execution

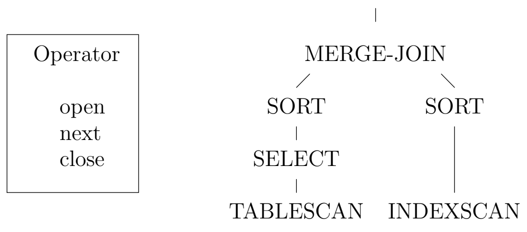

Figure 2.4.: Operator interface and usage example

is restricted to the page itself. Based upon this, complex *data structures* (e.g. relations or indices) are stored in the pages. They can span multiple pages, but have to use pages as storage units (e.g. each node in a B-tree would occupy exactly one page). The data structures offer a high level interface that hides the actual storage structure. Typical operations are insert a tuple, enumerate all tuples, etc. The topmost layer uses this high-level interface to *execute plans*. This is done by executing operators each performing a relative simple task (e.g. set intersection). They are combined to produce the final query result. As this is the layer most relevant for query processing, we describe it in more detail.

Algebraic operators are the building blocks of query execution. They offer a set-oriented (or bag-/list-oriented) view of the data. This means that each operator produces a set of data items (usually tuples) and itself takes sets of tuples as input. Semantic constraints aside operators can be combined arbitrarily, which makes them a very powerful concept for query execution. The basic operator interface and a usage example are shown in Figure 2.4. The standard operator interface [50] consists of the three methods *open*, *next* and *close*. The *open* method initializes or resets the operator, the *next* method returns the next data element until the whole result has been computed, and the *close* method finishes the computation and releases allocated resources. Note that this interface only cares about abstract data elements without understanding the actual contents of the data. For most operators, the actual data has only a limited influence on the operator logic. This allows for a generic implementation that is independent of the concrete data types. Therefore, required operations like comparisons are moved out of the operators and provided as an annotation to the operator. This allows writing generic operators that can be used for any kind of data.

To illustrate this concept, consider the query

```
select *
from person p,
     department d
where p.dep=d.id and
      p.salary>40000 and
      d.name="Development".
```

A possible operator tree for this query is shown in Figure 2.4: The `TABLESCAN` requires no input and returns as output the tuples contained in the `person` relation. This output is filtered by the `SELECT` operation, which removes all persons with a salary ≤ 40000 (not shown in the figure). The remaining tuples are passed to the `SORT` operator that reorders the tuples according to the `dep` attribute (not shown). The `INDEXSCAN` also requires no input and uses an index on the attribute `name` for the `department` relation. The index lookup implicitly evaluates the condition for `name`, and the output is passed to the other `SORT` operator that orders the tuples according to the `id` attribute. Both output streams are combined in the `MERGE-JOIN` operator, that combines tuple pair with matching `dep` and `id` attributes into a large tuple. The output of this operator consittutes the answer to the query.

Note that it is common to use an algebraic notation instead of operator names, e.g., $\bowtie$ instead of `JOIN`, σ instead of `SELECT` etc. When it is not clear which operator corresponds to the algebra expression (especially for joins), this is stated by using a superscript, e.g., $\bowtie^{SM}$ instead of `SORT-MERGE-JOIN`.

While support for DAG-structured query plans mainly involves the compile time system, the plan execution layer of the runtime system is also affected. For DAG-structured query plans, the operators can share operators as input. This makes it difficult to use the operator interface as described above. A detailed discussion of executing DAGs can be found in Chapter 11.

3. Related Work

Few papers about DAG structured query graphs exist and the techniques described there are usually very limited in scope. A Starburst paper mentions that DAG-structured query graphs would be nice, but too complex [31]. A later paper about the DB2 query optimizer [20] explains that DAG-structured query plans are created when considering views, but this solution uses buffering. Buffering means that the database system stores the intermediate result (in this case the view) in a temporary relation, either in main memory or on disk if the relation is too big. This buffering is expensive, either because it consumes precious main memory which could be used for other operators or – even worse – because the data has to be spooled to disk. Besides, DB2 optimizes the parts above and below the buffering independently, which can lead to suboptimal plans. Although not optimal, this is still a useful optimization and probably state of the art in commercial database management systems.

The Volcano query optimizer [21] can generate DAGs by partitioning data and executing an operator in parallel on the different data sets, merging the result afterwards. Similar techniques are described in [22], where algorithms like select, sort, and join are executed in parallel. However, these are very limited forms of DAGs, as they always use data partitioning (i.e., in fact, one tuple is always read by one operator) and sharing is only done within one logical operator.

Although few papers about the general construction of DAG-structured query plans exist, many published optimizations generate DAGs. A very nice optimization technique are so-called magic sets [5]. There, domain information from one part of the query graph is propagated sideways to another part of the query graph and used with a semijoin to remove tuples that can never match later in the query. A similar domain-based optimization can be used for dependent joins: instead of performing the dependent join for each tuple on the left-hand side, one determines the domain of the free variables on the right-hand side first, performs the dependent join only on the domain, and then joins the result with the left-hand side by means of a regular join. In both cases, the output of an operator is passed into two operators in separate parts of the query plan, so a more general DAG support is required.

Disjunctive queries can be handled efficiently via bypass plans [73]. There, tuples are passed to different operators depending on predicate tests. For example, when using the filter predicate $f(x) \lor g(x)$, the predicate $f(x)$ is evaluated first, and when it is satisfied the tuple can already be passed to the output without testing $g(x)$. As this can also be done in more complex situations, the performance gain can be substantial. This requires an even more general DAG support, as now operators can not only have multiple consumers but also pass tuples with different characteristics (especially cardinality) to their according consumers.

Sharing intermediate results is also important for multi-query optimization. One paper that constructs (limited) DAGs is [41]: It uses heuristics to identify common expressions in a sequence of (generated) queries and factorizes them into either a temporary view or a temporary relation. In an OLAP environment with generated queries, this reduced the runtime by up to a factor of 10.

While these techniques give a good impression of what kind of DAG support is required, many other techniques exists. Usually, the DAG requirements are simpler, the optimization techniques just want to read the same intermediate result multiple times (e.g. when optimizing XQuery expressions [29]). However, these papers never mention how this should actually be integrated into a plan generator and a runtime system. The plan generation problem can be avoided by using these techniques only as heuristics during the rewrite phases, but this can produce suboptimal results and is not really satisfactory. The runtime system is less important when considering only limited DAGs (and accepting the performance penalty caused by buffering intermediate results), but DAG support without buffering and with support for complex bypass plans is not trivial.

The only paper that explicitly handles DAG-structured query plans during query optimization is [67]. It describes some equivalences for operators with more than one consumer and then describes plan generation for DAGs. However, this is reduced to classical tree-structured query optimization: the algorithm decides beforehand (either using heuristics or by enumeration) which operators should be shared. Then, it duplicates the shared operators for all consumers except the first one and sets the costs for executing the duplicate to zero. Then it performs a tree-based plan generation and merges the duplicates afterwards. While this indeed generates DAG-structured query plans, one must be careful to avoid missing the optimal solution: first, the plan generator must be absolutely sure that the real operator (which represents the real costs) is included in the result plan and not only duplicates with zero costs. This can be a problem when constructing plans bottom-up and considering plan alternatives (i.e. subproblems can be solved by using different operators). Second, this only works if all operators just add up the costs of their input operators. When, for example, such a duplicate is present on the right-hand side of a nested loop join, the cost calculation will be completely wrong, as the nested loop multiplies the costs of the right-hand side (apparently zero here) with the number of tuples on the left-hand side. Still, it is a first step towards optimizing DAG-structured query plans. The paper does not handle the execution of these plans, it just mentions that they are useful for parallel/distributed execution.

4. Rule-based Query Compilation

4.1. Overview

The early query compilers like [69] were hard-wired, which means that every optimization performed was explicitly coded by hand. This severely limits the extensibility and also makes it more difficult to use certain meta heuristics like A^*. Extensibility is important for a query compiler, as adding extensions to the database system is quite common. Sometimes these extensions can even be made by the user (e.g. Starburst [31]), which requires a very flexible system. In particular, the following common modifications must be anticipated:

- adding support for new components of the runtime system, like new index structures, new operators etc.,
- adding support for new query features, like new query language standards, new data types etc.,
- adding new optimization techniques, like new algebraic equivalences, and
- reordering optimization steps, e.g., performing the most promising optimizations first.

In a hard-wired query compiler, this usually means rewriting large parts of the compiler. And that is not even an option if the query compiler should optimize user-defined types and functions, potentially with user-provided optimization techniques. Therefore, rule-based query compilers were developed that separated the query compiler from the concrete optimization steps, such that new optimizations could be added more easily. In the rest of this chapter, we first look at some existing systems and then discuss some design decisions for rule-based systems. A concrete implementation is discussed in Chapters 5 and 7.

4.2. Related Work

One of the first systems to use a rule-based approach was Squiral [71]. It uses a transformation-based optimization approach. The query is represented as an operator tree that is successively transformed by optimization rules. These rules are simple heuristics (e.g. perform a selection before a join) and are not cost-based. Still, the optimization can be extended by adding new rules. Besides, the query compiler used two rule sets. One for the main query transformation and afterwards one for order optimization to add sort operators for sort merge joins etc.

A much more ambitious approach was chosen by the Genesis project [6, 7]. The main goal there is to organize the query compiler (and the rest of the system) into reusable software components that could be combined as needed. It defines standard interfaces to get a clear separation between query representation, optimization algorithms and runtime algorithms. By using a generic representation all algorithms should be applicable, independent of the concrete database system. The optimizations are treated as transformation algorithms (they get an operator tree as input and produce a new tree), but could work constructively internally. While this is an interesting approach, it is not clear if this kind of generality can really be achieved in a query compiler. All optimization algorithms should work on any kind of input, but if a new operator is added the existing algorithms at least need some hints how to handle the operator (ignore it, optimize its input independent of the rest, etc.).

A well-known rule-based query compiler is the Exodus compiler generator [27]. It takes a specification of the logical algebra, the physical algebra, optimization rules and of rules to convert the logical algebra into the physical algebra. This specification is converted into source code that forms the actual query compiler. Extending the query compiler can be done by just changing the formal specification. The optimizations themselves are transformation rules that are applied in a cost-based manner using a hill-climbing model. To improve the search space exploration, the most promising rules (with the greatest estimated potential) are applied first. The Volcano query compiler [23, 28] is the successor project, that eliminates some limitations of the Exodus approach. These are mainly lack of support for parallel execution, a limited cost model and no support for enforcers (helper rules that guarantee ordering properties etc.). See Section 5.2 for a discussion of the plan generation approach of Volcano. The rule-based approach in general is similar to the Exodus project [28]. Based upon experiences with Exodus and Volcano, the Cascades framework [24] uses a more general rule set. The query optimizer no longer handles logical and physical operators differently (in fact, operators can be both) and it knows about enforcer rules. The Cascades framework is not as well published as the previous compilers, but apparently it is no longer a query compiler generator but just one query compiler. The rules no longer come from a formal specification but are now coded directly and integrated into the query compiler by providing an object-oriented interface.

A rule-based query compiler that does constructive instead of transformative plan generation is the Starburst query compiler [31, 44, 48]. It also has some transformation rules for predicate push-down etc., but these are only used during the rewrite phases. The plan generator itself builds the query plans bottom-up, combining LOLEPOPs (low-level plan operators) into more complex STARs (strategy alternative rules). The specification of this construction is done in a grammar-like way. The LOLEPOPs form the terminals and the STARs the non-terminals. This approach is especially interesting for two reasons. First, it is the only approach presented here that will construct the optimal solution (transformation-based optimization usually cannot guarantee this, as it does not consider the whole search space). Second, it allows the user to define new LOLEPOPs and STARs, so that the query optimizer can be extended by the user.

4.3. Design Decisions

When building a rule-based query optimizer, there are several decisions to be made about the rule interface. The first is what the rule should actually specify. The easiest possibility is to use rules as transformative optimizations, e.g., by specifying algebraic equivalences. This is done in most rule-based systems, but it limits the query optimizer to a transformative approach. To allow construction-based optimizations, the rules have to specify how operators can be combined. The best approach here is probably the Starburst model, which uses transformation rules during the rewrite phases and grammar-like rules for constructive plan generation. The plan generator presented here also uses this model, although the search is performed top-down: like in a top-down parser, the non-terminals are expanded until only terminals are left.

Another problem is the representation of the query execution plan (respectively the representation used for plan generation). To stay extensible, the query compiler should assume as little as possible about the concrete database system. However, the optimization rules need information about the query structure and potentially about the data types involved. There is no obvious solution to this problem. In a transformation-based optimizer the query could be represented as an algebraic expression which contains the full type information. But when a new operator is added to the system, the existing transformation rules do not know how to handle this operator. Probably they could ignore it and optimize the rest of the algebra expression, but getting a reasonable behavior might require to change all existing rules. Our plan generator uses a different approach. It is constructive and organizes the rules similarly to Starburst in a grammar-like way. However, the rules usually assume that all operators are freely reorderable. If this is not true, explicit operator dependencies are added during the preparation step (see Section 5.3). Thus, the plan generator treats evaluation order requirements like normal syntax requirements (attributes must exist etc.) that have to be checked anyway. The advantage of this method is that it is very flexible and assumes nearly nothing about the operators involved. The disadvantage is that the preparation step becomes more complex, as all equivalences have to be encoded in this step. Still, this is done only once per query, and, as a consequence, the checks during plan generation are much simpler.

Finally, the rules have to be encoded somehow. Most systems favor a formal specification that is either converted into source code (Exodus, Volcano etc.) or interpreted at runtime (Starburst for STARs). Other systems write the rules directly in source code (Starburst for transformation rules, Cascades). A formal specification is nice and might make transformation rules more efficient (as a generated tree parser can be more efficient than a hand written one), but it is difficult to make a specification expressive and extensible enough. The Cascades approach encodes all rules in objects that provide the required methods to the plan generator. This is a nice concept that was also chosen for our plan generator, although some care is required when creating these objects to keep them easily extensible. See Chapter 7 for a basic discussion of optimization rules written this way.

5. Extensible Approach for DAG Generation

5.1. Introduction

We present an extensible approach for the generation of DAG-structured query plans. As we will see in the next chapters, supporting DAG-structured query plans requires changes in different parts of a database system. The most prominent part, however, is the plan generator.

The plan generator is a central part of the query compiler. It takes a logical representation of the query and transforms it into a preferable efficient plan that can be executed by the runtime system of the database. Usually, this is done by converting an expression of a logical calculus or of a logical algebra into an expression of a physical algebra.

Since there are many different ways to express a logical expression as a physical one, the search space for the plan generator is very large. In fact, it can be shown that just determining the optimal join order is NP-hard in general [9, 35, 68]. While real-life queries can still be solved by using techniques like dynamic programming and pruning, the large search space results in very memory-intensive and computation-intensive operations. This requires some care when implementing a plan generator.

Another problem is the coupling between the plan generator and the rest of the system. This was already discussed in Chapter 4, however, the coupling is especially difficult when trying to minimize space requirements. To save space, the plan generator only retains the essential information in intermediate results and synchronizes with the rest of the system only at the beginning and at the end of the plan generation process (see Section 5.3).

In this chapter, we present a plan generator that makes very few assumptions about the actual database system and allows the efficient generation of DAG-structured query plans. Note that for practical purposes more information about the runtime system is required. This is modeled separately by the rules presented in Chapter 7.

The plan generator presented here handles DAG-structured query plans as efficiently as tree-structured query plans. In fact, generating DAGs is more efficient in some situations, as the plan generator recognizes equivalent subexpressions and prunes them against each other, thereby reducing the search space.

5.2. Related Work

The early papers about relational algebra already cared about optimizing algebra expressions [10, 64]. One of the first systems that used a proper plan generation as discussed in this chapter was System R [3, 69]. Its plan generator first determines the possible access paths (table scans or index scans) and then combines them bottom-up using dynamic programming until all relations are joined. While this only solves the join ordering problem, it is already a complex cost-based plan generation.

The Starburst plan generator [31, 44, 48] also uses a bottom-up construction similar to System R. However, instead of only considering joins it uses a rule-based approach to support arbitrary operators (see Section 4.2). Besides, it uses a more elaborated plan representation and a "glue" layer to enforce certain plan properties. The plan generator presented here uses a somewhat similar rule concept and similar partial plans, but the search phase is top-down and quite different.

The Volcano query compiler [23, 28] uses a transformation-based approach. It starts with a tree in logical algebra and performs a top-down search. In each step it either uses a transformation rule, converts a logical operator into the physical algebra, or adds an enforcer (e.g. *sort*) to guarantee physical properties. Memoization is used to reduce the search space. So in each step the plan generator receives a goal (a logical expression and some physical properties) that it tries to achieve. This concept of a goal-driven plan generation was reused in our plan generator, although in a constructive approach. The Cascades query compiler [24, 25] relaxes some of the constraints of the Volcano compiler. First, the rules are much more general, they can place enforcer, match entire subtrees etc. Second, the search space exploration is much more arbitrary. While Volcano used a top-down approach, Cascades explores the search space according to some guides, either explicit rules or promises made by rules. This means that the optimized expression can become an arbitrary mix of physical and logical operators, as the optimizer could optimize the parts in any order (in fact, Cascades makes nearly no difference between logical and physical operators). While this arbitrary exploration of the search space might have advantages (e.g. early plans), it is not clear if this could also be used for constructive plan generators for the following reason: when building plans constructively, the plan structure of subproblems is unknown until they have been solved. This makes optimization decisions dependent on these (yet unsolved) problems difficult.

A plan generator that explicitly handles DAGs is described in [67]. It uses a two-step approach that reduces the problem of generating DAGs to the problem of generating trees. In the first step, the query is analyzed and all operators that might be shared are determined. Then the subset of operators that should be shared is determined (either by exhaustive search or by using some heuristics, which might require running the following steps multiple times) and the shared operators are duplicated. The duplicates provide the same properties as the original operator, but report their costs as zero, so that additional consumers produce no costs. Then, a normal tree-based plan generation is performed and in the result the duplicates are merged back into the original operator. This

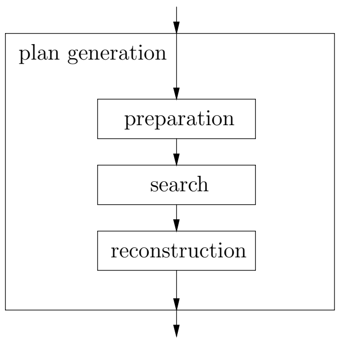

Figure 5.1.: Steps of plan generation

results in a cost-based DAG generation, but the approach has some drawbacks. First, it is unfortunate that the selection of shared operators has to be done beforehand, as this requires running the expensive plan generation step multiple times. This selection cannot be omitted easily (e.g. by assuming that all possible operators are shared), as some operators are alternatives (i.e. only one of these operators makes sense). For example, when considering materialized views, both a scan over the materialized view and a plan to recalculate the view could be used. If the view is read several times, these alternatives are both duplicated to enable sharing. By producing too many duplicates the plan generator will choose only the duplicates without the original operators, as they pretend to cause no costs. Second, the concept that additional consumers cause no costs is only valid if the plan generator does not consider nested loops. If data is read multiple times, the plan generator has to determine the maximum number of reads, and in the model described above the duplicates can be read an arbitrary number of times without causing costs. This severely underestimates costs, especially for very expensive operators. Note that nested loops cannot be completely avoided, both dependent joins and joins with non-equijoin predicates (e.g. `a like b`) use nested loops.

Another work that briefly mentions DAG generation is [55]. It gives some exemplary transformation rules where input can be shared and states that renames can be used to integrate the shared operators into the normal plan generation process. While this allows DAG generation, it was not used to build a query optimizer.

5.3. Integration of the Plan Generator

Before looking at the plan generator itself, it is worthwhile to consider the integration of the plan generator into the rest of the query compiler. As discussed in Section 2.2, in most systems the plan generator forms the boundary between primary logical and primary physical optimization. This means that the plan generation phase performs a significant change of representation, converting a logical calculus expression or algebra expression into a physical execution plan.

Plan generation itself can be separated in three distinct phases (see Figure 5.1): The first phase, *preparation*, takes the logical representation and brings it into a form suitable for plan generation. This includes looking up a lot of data that is relevant for the query and is too expensive to look up during the plan generation itself: relevant physical operators, available access paths, interesting orderings, data distribution statistics, selectivities etc.

The main plan generation step, *search*, takes these operators and tries to find the cheapest valid combination that is equivalent to the query. The exact way to do this differs between plan generators. Some just transform the operator tree, others build it bottom-up or top-down, but the search itself is basically a combinatorial problem. Since the search space is huge, these combinations have to be constructed as fast as possible. Therefore, the preparation phase should precompute as much data as possible to allow for fast tests for valid operator combinations, fast cost calculation etc. Furthermore, when using techniques like dynamic programming, the search phase can construct millions of partial plans, requiring a large amount of memory. Therefore, the search phase uses a different representation of the query that is optimized for the plan construction. The initial conversion from the query into this internal representation is also done by the preparation phase.

Finally, after the search phase has found the best plan, the *reconstruction* phase converts this plan back into the normal representation of the query compiler, although using physical operators instead of logical operators. While the preparation step can be involved, the conversion back is usually much simpler, the main problem is just to map the condensed internal representation back to the original logical operators and to annotate the corresponding physical operators accordingly.

When using a constructive plan generator instead of a transformative one, the preparation phase has an additional task: it has to determine the building blocks which will eventually be combined to form the full query. While this is simple when only considering table scans, it becomes much more complex when also considering index scans and materialized views [45]. If the preparation step misses a potential building block, the plan generator cannot find the optimal plan. On the other hand, the preparation step should add building blocks with care, as the search space increases exponentially with the number of building blocks. We will look at this particular problem in Section 5.7.1.

5.4. Algebraic Optimization

5.4.1. Overview

Query optimization, and especially plan generation, is based upon algebraic equivalences. The plan generator uses them either directly by transforming algebraic expressions into cheaper equivalent ones, or indirectly by constructing expressions that are equivalent to the query. For tree-structured query graphs many equivalences have been proposed (see e.g. [18, 52]), but some care is needed when reusing them for DAGs.

When only considering the join ordering problem, the joins are freely reorderable. This means that a join can be placed anywhere where its syntax

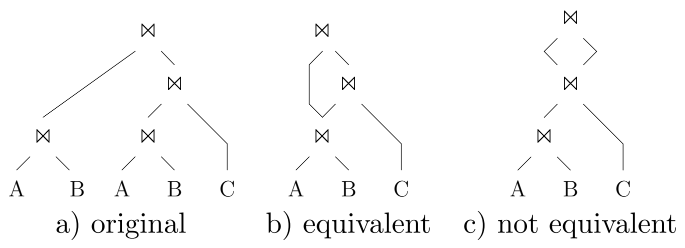

Figure 5.2.: Invalid transformation for DAGs

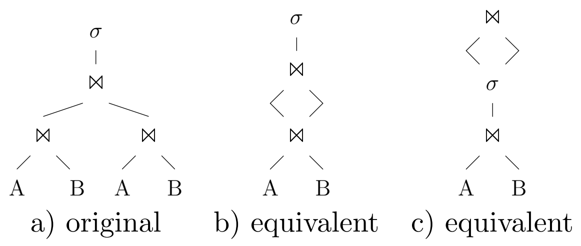

Figure 5.3.: Potentially valid transformation for DAGs (given a suitable σ)

constraints are satisfied (i.e. the join predicate can be evaluated). However, this is not true when partial results are shared. This is shown in Figure 5.2 for a query that computes the same logical expression twice (e.g. when using a view): In a) the join $A \bowtie B$ is evaluated twice and can be shared as shown in b). But the join with C must not be executed before the split, as shown in c), which may happen when using a constructive approach. Intuitively this is clear, as it means that $\bowtie C$ is executed on both branches. But in other situations a similar transformation is valid, as shown in Figure 5.3: There $A \bowtie B$ is also shared, then a self join is performed and a selection predicate applied to the result. Here, the selection can be executed before the topmost join if, for example, the selection considers only the join attributes (other cases are more complex). As the plan generator must not rely on intuition, we now describe a formal method to reason about DAG transformations.

The reason why the transformation in Figure 5.2 is invalid becomes clearer if we look at the variable bindings. As shown in Figure 5.4 a), the original

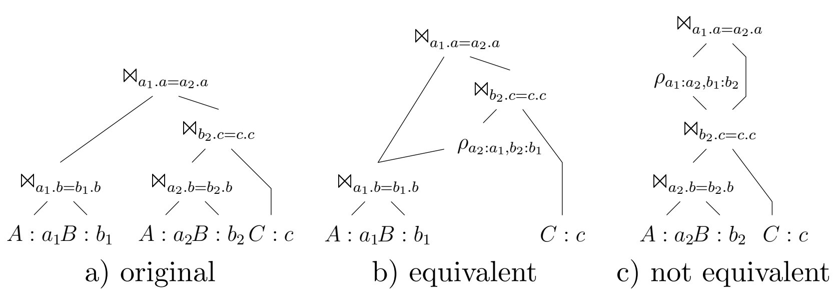

Figure 5.4.: More verbose representation of Figure 5.2

expression consists of two different joins $A \bowtie B$ with different bindings. The join can be shared in b) by renaming the output accordingly. While a similar rename can be used after the join $\bowtie C$ in c), this means that the topmost join joins C twice, which is different from the original expression.

This brings us to a rather surprising method to use normal algebra semantic: A binary operator must not construct a (*logical*) DAG. Here, logical means that the same algebra expression is executed on both sides of its input. What we do allow are *physical* DAGs, which means that we allow sharing operators to compute multiple logical expressions simultaneously. As a consequence, we only share operators by renames: If an operator has more than one consumer, all but one of these must be ρ operators. Thus, we use the ρ to pretend that the execution plan is a tree (which it is, logically) instead of the actual DAG.

5.4.2. Share Equivalence

Before going into more detail, we define when two algebra expressions are *share equivalent*, which means that one expression can be computed by using the other expression and renaming the result. We define

$$A \equiv_S B \text{ iff } \exists_{\delta_{A,B}:\mathcal{A}(A)\rightarrow\mathcal{A}(B)\, bijective}\, \rho_{\delta_{A,B}}(A) = B.$$

As this condition is difficult to test in general, we use a constructive definition for the rest of this work (which in fact consists of sufficient conditions for the definition above). First, two scans of the same relation are share equivalent, as they produce exactly the same output (with different variable bindings). Note that in this constructive approach the mapping function $\delta_{A,B}$ is unique. Therefore, we always know how attributes are mapped.

$$scan_1(R) \equiv_S scan_2(R)$$

Other operators are share equivalent if their input is share equivalent and their predicates can be rewritten using the mapping function. For the operators used in this work (see Appendix A) we use the following definitions:

$$\begin{array}{rcll}
A \cup B & \equiv_S & C \cup D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \cap B & \equiv_S & C \cap D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \setminus B & \equiv_S & C \setminus D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
\Pi_A(B) & \equiv_S & \Pi_C(D) & \text{if } B \equiv_S D \wedge \delta_{B,D}(A) = C \\
\rho_{a\rightarrow b}(A) & \equiv_S & \rho_{c\rightarrow d}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = c \wedge \delta_{A,B}(b) = d \\
\chi_{a:f}(A) & \equiv_S & \chi_{b:g}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = b \wedge \delta_{A,B}(f) = g \\
\sigma_{a=b}(A) & \equiv_S & \sigma_{c=d}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = c \wedge \delta_{A,B}(b) = d \\
A \times B & \equiv_S & C \times D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \bowtie_{a=b} (B) & \equiv_S & C \bowtie_{c=d} (D) & \text{if } A \equiv_S C \wedge B \equiv_S D \wedge \delta_{A,C}(a) = c \wedge \delta_{B,D}(b) = d \\
A ⟕_{a=b} (B) & \equiv_S & C ⟕_{c=d} (D) & \text{if } A \equiv_S C \wedge B \equiv_S D \wedge \delta_{A,C}(a) = c \wedge \delta_{B,D}(b) = d \\
A \ltimes_{a=b} (B) & \equiv_S & C \ltimes_{c=d} (D) & \text{if } A \equiv_S C \wedge B \equiv_S D \wedge \delta_{A,C}(a) = c \wedge \delta_{B,D}(b) = d \\
A \overrightarrow{\bowtie}_{a=b} (B) & \equiv_S & C \overrightarrow{\bowtie}_{c=d} (D) & \text{if } A \equiv_S C \wedge B \equiv_S D \wedge \delta_{A,C}(a) = c \wedge \delta_{B,D}(b) = d \\
\Gamma_{A;a:f}(B) & \equiv_S & \Gamma_{C;b:g}(D) & \text{if } B \equiv_S D \wedge \delta_{B,D}(A) = C \wedge \delta_{B,D}(a) = b \wedge \delta_{B,D}(f) = g \\
\mu_{a:b}(A) & \equiv_S & \mu_{c:d}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = c \wedge \delta_{A,B}(b) = d
\end{array}$$

These conditions are much easier to check, especially when constructing plans bottom-up (as this follows the definition).

Note that the share equivalence as calculated by the tests above is orthogonal to normal expression equivalence. For example, $\sigma_1(\sigma_2(R))$ and $\sigma_2(\sigma_1(R))$ are equivalent (ignoring costs), but not share equivalent when only testing the sufficient conditions (this is not a problem for plan generation, as the plan generator considers both orderings). On the other hand, $scan_1(R)$ and $scan_2(R)$ are share equivalent, but not equivalent, as they produce different attribute bindings. Share equivalence is only used to detect if exactly the same operations occur twice in a plan and, therefore, only once cause costs (ignoring nested loops, see Chapter 9 for more details). The logical equivalence of expressions is handled by the plan generator anyway, it is not DAG-specific.

Using this notion, the problem in Figure 5.2 becomes clear: In part b) the expression $A \bowtie B$ is shared, which is ok, as $(A \bowtie B) \equiv_S (A \bowtie B)$. But in part c) the top-most join tries to also share the join with C, which is not ok, as $(A \bowtie B) \not\equiv_S ((A \bowtie B) \bowtie C)$.

5.4.3. Optimizing DAGs

The easiest way to reuse existing equivalences is to hide the DAG structure completely: During query optimization the query graph is represented as a tree, and only when determining the costs of a tree the share equivalent parts are determined and the costs adjusted accordingly. Only after the query optimization phase the query is converted into a DAG by merging share equivalent parts. While this reduces the changes required for DAG support to a minimum, it makes the cost function very expensive. Besides, when the query graph is already DAG-structured to begin with (e.g. for bypass plans) the corresponding tree-structured representation is much larger (e.g. exponentially for bypass plans), enlarging the search space accordingly.

A more general optimization can be done by sharing operators via ρ operators. While somewhat difficult to do in a transformation-based query optimizer, for a construction-based query compiler it is easy to choose a share equivalent alternative and add a ρ as needed. Logically, the resulting plans behave as if the version without ρ was executed (i.e. as if the plan was a tree instead of a DAG). Therefore the regular algebraic equivalences can be used for optimization. We will look at this again when discussing the plan generator.

5.4.4. Optimal Substructure

Optimization techniques like dynamic programming and memoization rely on an optimal substructure of a problem (neglecting properties). This means that the optimal solution can be found by combining optimal solutions for subproblems. This is true when generating tree-structured query graphs, but when done naively, is not true for DAGs. Figure 5.5 shows two query graphs for $A \bowtie B \bowtie C \bowtie B \bowtie C \bowtie D$. The graph on the left-hand side was constructed bottom-up, relying on the optimal substructure. Thus, $A \bowtie B \bowtie C$ was optimized, resulting in the optimal join ordering $(A \bowtie B) \bowtie C$. Besides, the optimal

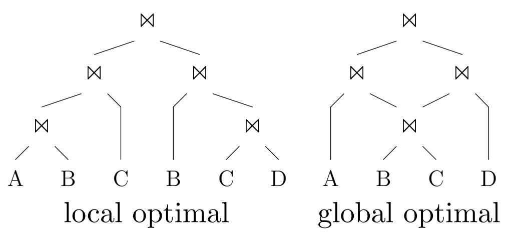

Figure 5.5.: Possible non-optimal substructure for DAGs

solution for $B \bowtie C \bowtie D$ was constructed, resulting in $B \bowtie (C \bowtie D)$. But when these two optimal partial solutions are combined, no partial result can be reused (except the scans of B and C, but these were omitted due to clarity reasons). When choosing the suboptimal partial solutions $A \bowtie (B \bowtie C)$ and $(B \bowtie C) \bowtie D$, the expression $B \bowtie C$ can be shared, which might be a better plan. Therefore, the optimal DAG cannot be constructed by just combining optimal partial solutions.

This could be solved by deciding beforehand which operators should be shared. Then the plan generator would prefer plans that allow sharing these operators (i.e. where the operators form a subgraph) to an otherwise equivalent plan that does not allow sharing the operators. As it is not possible in general to decide which operators should be shared, the plan generation works slightly different: Instead of creating plans which allow sharing a given set of operators, each plan is annotated with the set of operators in the plan that could be shared with other plans. In theory, this would be every operator in the plan, but in practice, only a few operators are relevant for sharing. We look at this in more detail in Section 6.3.1. Given this annotation, the plan generator can check if one plan allows more sharing than the other, and keep it even if it is more expensive. As this means one plan per set of shared operators, the search space is increased by a factor of 2^n, where n is the number of sharable operators. In practice, the factor is much lower, usually proportional to n and often close to 1. This has several reasons.

First, 2^n is a very loose upper bound, as only operators can be shared that are actually part of the subproblem. This still results in an exponential growth of the factor, although much slower. Besides, only operators can be shared whose input is also shared, as we want to share whole subgraphs. Furthermore, the plan generator can still prune dominated plans. So when one plan allows sharing the same or more operators than another plan and has lower costs, it dominates this plan. Therefore, it is often sufficient to keep one plan if it offers the most sharing and, at the same time, has the lowest costs. Besides, it is possible to estimate if sharing is worthwhile: The costs for the plan without sharing in Figure 5.5 are at most twice as high as for the plan with sharing, as it selected the (local) optimal join ordering, and its only fault is calculating $B \bowtie C$ twice. Therefore, the join ordering $A \bowtie (B \bowtie C)$ can be discarded if it causes more than twice the costs of the optimal solution, as the difference cannot be made up. In general, plans with a greater sharing can be discarded if the costs are greater than the costs of the optimal solution times the maximum sharing (the maximal number of shared plans).

While the plan generator presented here guarantees the optimal solution by keeping track of shared operators, sharing could be excluded from the search space as a heuristic. As stated above, the maximum error made by constructing the optimal solution without taking future sharing into account is bound by the maximum sharing, usually 2 or 3. Besides, common subexpressions are still shared if possible. It may only happen that some sharing opportunities are missing. Cases where it is better to use suboptimal partial results can be constructed, but do not happen for typical queries (e.g. TPC-H). They require that the partial solution with more sharing is not much worse than the optimal solution (otherwise sharing does not pay off), and only differs in the operator order. This is uncommon.

5.5. Approach

5.5.1. Goals

The main design goals of the plan generator presented in this chapter were:

1. support for DAG-structured query plans
2. support for arbitrary operators
3. support for materialized views
4. support for bypass plans
5. efficient plan generation

Of these goals, efficiency was the least important. Of course the plan generator still had to be able to optimize large queries, but a certain slowdown compared with a heavily optimized (classical) plan generator was acceptable if the other goals were met.

Support for a wide range of operators and materialized views is now common for plan generators, however, support for DAG structured query plans is not. Therefore, the main goal was to support DAGs, although the other goals are also achieved with the chosen approach.

5.5.2. Components of the Plan Generator

When talking about "the plan generator", one usually means "the component that constructs the physical plan". However, the plan generator is split into several components where the plan construction itself is comparatively small, as a lot of infrastructure is required during the plan generation phase. A rough overview of the components used in our approach is shown in Figure 5.6. They are discussed in more detail in the following chapters, but to make the concept of the plan generator clearer, we already give a short overview here.

The core component of the plan generator is the *search component*. It is responsible for finding the best plan equivalent to the input and uses the other

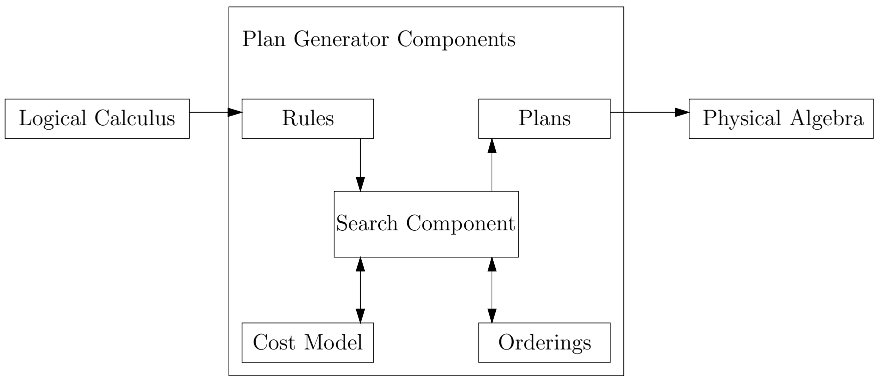

Figure 5.6.: Components of the Plan Generator

components as helpers. In our approach, it is a kind of meta algorithm, which means that the optimizer knows how the best plan should be found, but leaves all the details to other components. This has the advantage that the optimizer is very generic and extensible. We will discuss this in more detail in the rest of the chapter.

The semantic coupling between the optimizer and the rest of the system is done by the *rules* component. In the preparation phase the input is analyzed and instances of all rules relevant for the query are instantiated. A rule can, e.g., correspond to one algebra operator. So when the query contains a `where` condition like `a=b`, a rule instance for a selection (or a join) is created that contains information like required attributes, selectivity and how to construct a physical operator from the result of the search phase. As we will see in Chapter 7, there is no simple 1:1 mapping between rules and operators, but for now it is sufficient to think of the rule instances as descriptions for all physical operators that might be useful to answer the current query.

The plan generator builds the final plan incrementally, which means that it creates many partial plans which only answer a subproblem of the input query. The plan management is done by the *plans* component, which also takes care of pruning plans that are dominated by other plans.

Finally, the *cost model* and the reasoning about *orderings* and groupings are organized in dedicated components, as the corresponding problems are complex and can be clearly separated from the rest of the plan generator. See Chapters 8 and 9 for a detailed description.

5.5.3. An Abstract View of Plan Generation

When looking at the problem of plan generation from a very high level, a plan generator is a mechanism that finds the cheapest plans with some given properties.

Consider the SQL query and its canonical operator tree shown in Figure 5.7. The plan generator has to find the cheapest variation of this tree that still returns the same result. Here, the only beneficial change to the operator tree

```
select c.name, sum(o.value)
from customers c,
     orders o,
where c.id=o.customer and
      o.year=2004
group by c.id, c.name
```

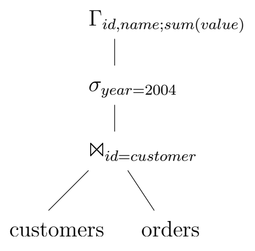

Figure 5.7.: A SQL query and its canonical operator tree

would probably be pushing the selection down. However, the plan generator also has to replace all logical operators by physical operators. In the example, here the join has to be replaced by a concrete join implementation. For a sort merge join, this might even add additional sort operators to the tree. So in this example, the plan generator has to find the cheapest combination of physical operators that computes the (logical) join, the selection and the group-by while still maintaining the proper semantics (e.g. the selection has to be done before the group-by).

In general, the plan generation phase can be formulated as a problem of satisfying (binary) properties under cost constraints: The plan generator tries to find the cheapest plan that "includes" all logical operators requested by the input. Of course, the plan consists of physical operators, but these operators know which logical operators they represent (as we will see in Chapter 7, a physical operator can actually represent zero, one or many logical operators). The operators themselves also require certain properties, the simplest ones are available attributes: In the example, the selection requires the attribute `year`, the join requires the attributes `id` and `customer` and the group-by requires the attributes `name` and `value`. However, looking at the attributes is not sufficient when considering operators that are not freely reorderable (like outer join or group-by). This is discussed in more detail in Chapter 7, but basically the constraints can also be modelled as tests for binary properties. In the following, we call these binary properties used to model operator requirements *bit properties* to distinguish them from other plan properties.

For the plan generator, the semantics of these bit properties are not really relevant, all it cares about is that these bit properties form the search space (alternatives with the same bit properties can be pruned against each other) and that they are required and produced by certain "operators". Actually, the plan generator does not even care about operators, it only reasons about rule instances, which roughly correspond to physical operators. So the plan generator uses the rules to combine partial plans until finally a plan is found that satisfies all bit properties required to answer the query. Of course this is more complex, as the plan generator tries to find the cheapest plan and that with as few computations as possible, but before looking at the details we first consider the rules used during the search.

5.5.4. A Concrete Plan Generator

The last paragraph actually describes a way to implement a plan generator. Using this model, plan generation consists of recursively finding plans with certain bit properties. It starts by searching a plan that contains all logical operators. In each step, it tries to use one of the known rules to produce some required bit properties and recursively finds plans with the bit properties that are still missing. While a bottom-up implementation is also possible, a top-down approach has the advantage that only bit property combinations are examined that are relevant for the total query (see e.g. Section 5.9). The basic approach is as follows:

```
PLANGEN($goal$)
1  $bestPlan \leftarrow$ NIL
2  for each $r$ in known rule instances
3    do if $r$ can produce a bit property in $goal$
4          then $rem \leftarrow goal \setminus \{p|p$ is produced by $r\}$
5               $part \leftarrow$ PLANGEN($rem$)
6               $p \leftarrow$ BUILDPLAN($r, part$)
7               if $bestPlan =$ NIL or $p$ is cheaper than $bestPlan$
8                 then $bestPlan \leftarrow p$
9  return $bestPlan$
```

This is highly simplified (see Section 5.7 for the real algorithm), but the basic idea is that the plan generator is only concerned with fulfilling abstract bit properties without understanding their semantics. The actual semantics are described by the rules, which also take care of commutativity, associativity etc. Because of this, the BUILDPLAN function used above might intentionally fail, as, although the plan generator has determined that all required bit properties are satisfied, the rules themselves might decide that the result would not be equivalent to a part of the query. In the real implementation, this is no problem, as the rules also influence the search space exploration and, therefore, can avoid this situation.

As the plan generation does not need to understand the bit properties and is only interested in the fact whether they are available or not, these can be modeled efficiently using bitmaps; the plan generation function simply maps a bitmap representation of the available bit properties to a partial plan. This observation also shows an obvious method to reduce the search space: As plans with a certain set of bit properties will be requested repeatedly, memoization can be used to avoid recomputation.

A nice property of the plan generator sketched above is that it creates DAGs automatically: Each operator is linked to a partial plan that satisfies certain bit properties. If two operators require plans with the same bit properties, this results in a DAG. For more aggressive sharing, some additional care is required (e.g. it might be possible to share a view that offers more attributes than required by each individual consumer), but in general, this avoids duplicate computation.

Support for materialized views is also very simple: Treat the materialized view like a scan that immediately provides all bit properties of the materialized view. This way, it is automatically included in the search process without any special handling. Consider, for example, the following query (assume that matview is materialized):

```
create view matview
   select   o.custkey, o.year, revenue:sum(i.revenue)
   from     order o, item i
   where    o.orderkey=i.order
   group by o.custkey, o.year

select c.name, v.revenue
from   customer c, matview v
where  c.custkey=v.custkey and
       v.year=1995
```

The plan generator has two choices: It either uses the materialized view, reducing the problem to two scans, one join and one selection, or it can ignore the materialized view and calculate the view again, which means three scans, two joins, one group-by and one selection. When only looking at the operators involved, the materialized view seems to be a clear winner (and in this simple example it probably is), but using the materialized view has the disadvantage that the selection predicate cannot be pushed inside the view. If the view is very large and the predicate is very selective, it might be better to ignore the materialized view. The decision can (and should) be made by the plan generator: First, look up all rules required to answer the query without using the materialized view during the preparation step. Then, add another rule that represents a scan of the materialized view and sets all bit properties as if the join and the group-by of the view had been executed, but with the cost characteristics of the scan. Now the plan generator is free to choose between the two alternatives and can select the cheaper one.

5.6. Plan Properties

5.6.1. General

During plan generation, the plan generator generates millions of partial plans that are finally combined to the full plan. Since these plans consume a significant amount of memory, a compact representation is essential. Choosing a representation includes a time-space trade-off; e.g., the costs of a plan could be recomputed every time they are required. However, they are usually materialized, as a recomputation would be too expensive.

Traditionally, the plan properties include the cardinality, the costs, available attributes (including the tuple width), applied operators, the ordering, available indices and storage information (the site and materialization information) [48]. However, for the plan generator described here, the plan properties are much more abstract, consisting mainly of a bit set and some information for the cost model. We call these properties stored in a bit set *bit properties*, to distinguish them from plan properties in general. As the plan generator does not understand the semantics of the bit properties, it just keeps track of them and verifies that the equally abstract requirements of the operators are met. Actually, the set of stored bit properties is selected for the current query, as we will see in Section 5.7: The plan generator minimizes the set of bit properties during the preparation step and only keeps those that are essential for the current query. The set of potential bit properties is discussed in Section 5.6.2.

The plan generator tries to find the cheapest execution plan for the query and does so by first finding the cheapest (partial) plan for subproblems and then combining the plans to the complete execution plan. Ideally, there exists one "best" plan for each subproblem, but sometimes it is not clear which plan is better for one of either following reasons: because of the cost function (see Section 9.3.2), because of ordering/grouping properties or because some plans allow more sharing. For this reason the plan generator maintains a set of plans for each (relevant) bit property combination which contains all plans that satisfy these bit properties and are not dominated by other plans in the set. Note that ordering/grouping properties are handled separately: Theoretically the ordering and grouping properties could be treated, like any bit property, as a simple flag: is ordered on attribute a, is grouped by the attributes b and c etc. However, the set of all possible orderings is quite large and orderings can change transitively into each other, which would make maintaining such flags very difficult. Therefore, we use a special data structure for orderings, which is described in Chapter 8. This data structure is opaque for the plan generator, it only allows to check if a certain ordering or grouping is satisfied by the partial plan.

As the plan generator has to keep multiple plans for a subproblem, the plan management is split into two components: The container (*PlanSet*, which corresponds to one entry in the memoization table) describes the subproblem, contains all plans and holds all logical properties shared by the contained plans. The plan itself (*Plan*) describes a concrete plan and holds the physical properties, that are different for each plan. The concrete data structures are discussed in Section 6.1.

5.6.2. Properties for Operator Rules

While talking about the set of bit properties satisfied by a plan, we did not specify what these bit properties actually are. For the plan generator itself this is not relevant, as only the rules themselves care about the semantics of the bit properties. A detailed discussion of different rules can be found in Chapter 7, but we already present some simplified rules and their bit properties here to make the algorithm part of this chapter clearer.

We assume for a moment that we only want to optimize table scans, selections and joins. This can be done by using three different rules, which at the same time represent three physical operators: scan, selection and nested loop join. A scan requires no input and produces a tuple stream with a given set of attributes. In this scenario a selection is always applicable if the required attributes are present in the input and the join requires that the input streams contains certain relations (i.e. the attributes of these relations). In this simple model, the set of potential bit properties could just be the set of attributes relevant for the operators. This is sufficient for each rule to check if their corresponding operator would be applicable, however, it is not enough for the plan generator as a whole: When just looking at the attributes, the plan generator would try to apply selections multiple times, as they seem to reduce the cardinality ad infinitum. Therefore the set of bit properties should also contain the logical operations applied so far. Note that this can be different from the set of rules: there can be multiple rules that produce the same bit properties (e.g. for different join implementations). These rules would perform the same logical operator and, therefore, only use one bit property entry. In our simple scenario, one rule is equivalent to one logical operation.

Using attributes and logical operations as bit properties is very expressive, as dependencies between logical operations (e.g. an outer join and its input) can be modeled as well: An operation that has to be performed after another operation just requires that the other operation has to be executed somewhere in its subplans (which can be seen by looking at the bit properties). However, it is desirable to keep the set of bit properties small, as this allows a more efficient representation. Therefore, all rules selected during the preparation phase first register all bit properties that might be relevant, and then a pruning algorithm decides which bit properties should actually be used. For example, it is often not required to keep track of the fact that a join has been performed, as the join could not be executed twice anyway (due to the nature of search space exploration). We will look at this in Section 5.7.1.

Apart from the plan information shown above, a rule needs additional information to calculate the properties of a new plan. For example, a rule for a selection needs to know its selectivity, or a tablescan the physical characteristics of the table. This information is not stored in the plan itself, but in the (shared) rules: For each logical operation of the query (see Section 7.2 for more details), the preparation steps creates one (or more) new rule instance that contains all statistics and other parameters required by the rule. This has the advantage of keeping the plans compact while still maintaining the full information.

5.6.3. Sharing Properties

Apart from the bit properties used for the operator rules, each plan contains a *sharing* bit set to indicate potentially shared operators. Semantically, this belongs to the cost model, as it is used to detect DAG creation and to keep plans that allow more sharing. But as it is related to the algebraic optimization discussed in Section 5.4, we already discuss it here.

When considering a set of share equivalent plans, it is sufficient to keep one representative, as the other plans can be constructed by using the representative and adding a rename (note that all share equivalent plans have the same costs). Analogously, the plan generator determines all rules that are share equivalent (more precisely: could produce share equivalent plans if their subproblems had share equivalent solutions) and places them in equivalence classes (see Section 6.3.1). As a consequence, two plans can only be share equivalent if their producing rules are in the same equivalence class, which makes detecting share equivalence easier. Equivalence classes containing only a single rule are discarded, as they do not affect plan sharing. For the remaining equivalence classes, one representative is selected and one bit in the *sharing* bit set is assigned to it.

For example the query in Figure 5.5 consists of 11 operators: A, B_1, C_1, B_2, C_2, D, $\bowtie_1$ (A and B_1), $\bowtie_2$ (between B_1 and C_1), $\bowtie_3$ (between B_2 and C_2), $\bowtie_4$ (between C_2 and D) and $\bowtie_5$ (between A and D). Then three equivalence classes with more than one element can be constructed: $B_1 \equiv_S B_2$, $C_1 \equiv_S C_2$ and $\bowtie_2 \equiv_S \bowtie_3$. We assume that the operator with the smallest subscript was chosen as representative for each equivalence class. Then, the plan generator would set the *sharing* bit B_1 for the plan B_1, but not for the plan B_2. The plan $A \bowtie_1 (B_1 \bowtie_2 C_1)$ would set sharing bits for B_1, C_1 and $\bowtie_2$, as the subplan can be shared, while the plan $(A \bowtie_1 B_1) \bowtie_2 C_1$ would only set the bits B_1 and C_1, as the join cannot be shared (only whole subgraphs can be shared). The sharing bit sets allow the plan generator to detect that the first plan is not dominated by the second plans, as the first plan allows more sharing. The equivalence classes are also used for another purpose: When a rule requests a plan with the bit properties produced by an operator, the plan generator first checks if a share equivalent equivalence class representative exists. For example, if a rule requests a plan with B_2, C_2 and $\bowtie_3$, the plan generator first tries to build a plan with B_1, C_1 and $\bowtie_2$, as these are the representatives. If this substitution is possible (i.e. a plan could be constructed), the plan constructed this way is also considered a possible solution.

In general, the plan generator uses the sharing bits to mark sharing opportunities: Whenever a partial plan is built using an equivalence class representative, the corresponding bit is set, which means that the plan offers to share this operator. Note that it is sufficient to identify the selected representative, as all other operators in the equivalence class can be built by just using the representative and renaming the output. As sharing is only possible for whole subplans, the bit must only be set if the input is also sharable. Given, for example, three selections σ_1, σ_2 and σ_3, with $\sigma_1(R) \equiv_S \sigma_2(R)$. The two operator rules for σ_1 and σ_2 are in the same equivalence class, we assume that σ_1 was selected as representative. Now the plan $\sigma_1(R)$ is marked as "shares σ_1", as it can be used instead of $\sigma_2(R)$. The same is done for $\sigma_3(\sigma_1(R))$, as it can be used instead of $\sigma_3(\sigma_2(R))$. But for the plan $\sigma_1(\sigma_3(R))$ the *sharing* attribute is empty, as σ_1 cannot be shared (since σ_3 cannot be shared). The plans containing σ_2 do not set the *sharing* property, as σ_1 was selected as representative and, therefore, σ_2 is never shared.

The explicit marking of sharing opportunities has two purposes: First, it is required to guarantee the optimal results, as one plan only dominates another if it is cheaper and offers at least the same sharing opportunities. Second, it is useful for the cost model, as it has to identify the places where a DAG is formed (i.e. the input overlaps). This can now be done by checking for overlapping *sharing* properties. It is not sufficient to check if the normal bit properties overlap, as the plans pretend to perform different operations (which

1. instantiate suitable operator rules
2. register operator properties
3. prune irrelevant operators
4. minimize the properties
5. construct search filter
6. register share equivalent operators
7. register relevant orderings and groupings

Figure 5.8.: Steps of the preparation phase

```
select p.name,       select p.name, d.name  1. scan(persons)
       p.dep.name    from   persons p,      2. map(p.dep.name)
from   persons p            departments d   3. scan(departments)
                     where  d=p.dep         4. join(d=p.dep)
```

Figure 5.9.: A example query, an alternative formulation and relevant rules

they do, logically), but share physical operators.

5.7. Algorithms

After describing the general approach for plan generation, we now present the actual algorithms used. Unfortunately, the plan generator is not a single, concise algorithm but a set of algorithms that are slightly interwoven with the operator rules. This is unavoidable, as the plan generator should be as generic as possible and, therefore, does not understand the semantics of the operators. However, there is still a clear functional separation between the different modules: The plan generator itself maintains the partial plans, manages the memoization and organizes the search. The operator rules only describe the semantics of the operators and guide the search with their requirements.

In the following, we mainly describe the plan generator itself and only sketch the operator rules to illustrate the interaction with the plan generator; the rules are discussed in detail in Chapter 7. In this section, we first look at the coupling between the rules and the plan generator and then follow the plan generation step, starting with the preparation step, then the search phase and finally the plan reconstruction. Note that the discussion here is very high-level, a more detailed description of the algorithms is included in Chapter 6.

5.7.1. Preparation Phase

The different steps of the preparation phase are shown in Figure 5.8. The first step consists of traversing the input query and instantiating suitable rules. Which rules are actually suitable depends on the query and the concrete database instance, but it should include at least rules for the operators in the input, rules for available indices and rules for materialized views that could be used. If plan alternatives are considered (e.g. joins instead of pointer chasing), all rules for the operators of the plan alternatives must also be instantiated. All these rules are instantiated, annotated with information like selectivities etc., and collected.

Consider the OQL query shown on the left of Figure 5.9: It determines the name of the department for each person by pointer chasing. This can be evaluated by a single scan of *persons* and one map operator that does the pointer chasing, so that the rules 1 and 2 shown on the right are instantiated. If the query compiler realizes that the query in the middle is an equivalent formulation that eliminates pointer chasing by a join, the rules 3 and 4 are also instantiated. Rules 1 and 3 are annotated with the characteristics of the extents (cardinality, pages, tuples per page, tuple size, physical storage). Rule 2 needs the same statistics as rule 3 to estimate the costs of pointer chasing, and rule 4 needs a selectivity estimation and the information that the join is a $1 : n$ join. Note that the four rules are the rules directly instantiated in the preparation step. However, more rules are instantiated indirectly, e.g., the join rule will consider different join algorithms with different rules (see Section 7.5.4). While it is relevant to keep this in mind to understand the algorithm, these hidden rules are invisible for the plan generator which only reasons about the directly instantiated rules.

In the next step, the operator rules register their required and produced bit properties. Anything that can be expressed as a boolean expression can be a bit property, but the usual bit properties are available attributes and applied operators. In our example, the scan rules produce the attributes of their relations and the property that they have been applied. The map rule requires the attribute *persons.dep* and produces the attribute *departments.name* and the fact that it has been applied. The join rule requires the attributes *persons.dep* and *departments.oid* and produces the fact that it has been applied. Note that the bit properties "map applied" and "join applied" should be identified (by the rewrite rule that generated the two formulations), as they are plan alternatives.

After the bit properties are registered, the preparation algorithm determines which rules are relevant for the query. This step is redundant if it is always certain that a rule is relevant for the query, but when considering plan alternatives, it can easily happen that a rule looks promising but in the end cannot be used in a complete plan. Consider e.g. access support relations (ASRs, [38]) or materialized views that answer parts of the query. When they are used conservatively (i.e. they exactly match a part of the query), they are always usable, but when trying to use ASRs or materialized views that only overlap with a part of the query it can, but need not be possible to re-use this for the whole query. Of course the ultimate test for this is the plan generation itself, but to reduce the search space the plan generator tries to eliminate non-applicable rules beforehand. This is done by two different heuristics: First, all rules are removed which require a bit property that is never produced. Second, all rules are removed that are never required for the final plan, not even transitively. Note that these removals have to be performed at the same time: a rule might have impossible requirements whose removal makes another rule unnecessary whose removal makes a third rule impossible etc. As stated above, this step

rule	filter
1	*person.name/dep*
2	*map/join*, *deparment.name*, *person.name/dep*
3	*department.oid*, *department.name*
4	*map/join*, *deparment.name*, *person.name/dep*

Figure 5.10.: Filter for the rules in Figure 5.9

is not strictly required, but it makes considering plan alternatives (e.g. materialized views) simpler: All alternatives that answer some part of the query are considered; alternatives where it is certain that they cannot be used for the final solution are discarded by this step to reduce the search space.

After it is clear which rules will be considered during plan generation, the set of bit properties is minimized. Bit properties that are produced but never required are eliminated, and bit properties that are always produced simultaneously are merged. Minimizing the set of bit properties has two purposes: First, it reduces the space of the bitmap required in each plan and second, it reduces the search space, as now some bit properties are ignored which otherwise could have prevented the comparison of two plans. In our running example, this removes all bit properties for unused attributes and the *scan applied* properties, as only the attributes and not the scans themselves are required in this example (an implementation would probably also require the scans to avoid special cases, but then the *scan applied* property could be merged with one of the attribute properties). The *map/join applied* property is kept, as it is used by the map/join rules to detect already applied rules. The *person.name* and *person.dep* properties are merged, as they are only produced simultaneously. The final set consists of four bit properties, *person.name/dep*, *department.name*, *department.oid* and *map/join applied*. Technically, it would also be possible to eliminate *department.oid* (by merging rules 3 and 4 in one macro rule), but this would hide the scan of department from the plan generator and prevent a re-usage of the scan for more complex queries.

The minimized bit properties are now used to construct a search filter as a fast test if a plan with a given set of bit properties might actually be constructed by a certain rule. This is usually straight-forward, e.g., for most rules the filter is constructed as *requirements* ∪ *produced*. This means that a plan including the rule must at least have the bit properties required for the rule and the bit properties produced by the rule. The search filters for our example are shown in Figure 5.10. The filter for rule 4 (*join(d=p.dep)*) consists of *person.name/dep*, *department.name* and *map/join applied*, which is somewhat surprising: The property *department.oid* is included in the requirements but not in the filter. In a normal join situation it would also be included in the filter, but here we want to use the join as a plan alternative and, therefore, intentionally forget the additional available attributes. Note that it is safe to use any subset of these filters (in particular, each filter can be empty), the filters are only a performance optimization to check if a rule makes sense in a certain situation.

To support operator sharing, the plan generator now determines the equivalence classes for share equivalent operators (see Section 5.6.3). In each class one representative is selected and a bit in the *sharing* bit set is assigned to it. In our example, no bits are assigned, as all equivalence classes contain just one element (i.e. no sharing is possible). See Section 5.8 for a more complex example including operator sharing.

In the final preparation step the rules register all orderings and groupings that are relevant for them. This is described in more detail in Chapter 8. In our example the join would register (*person.dep*) and (*department.oid*), as orderings (useful for a sort merge join) and the map operator might register (*person.dep*) as a grouping, as it would make sense to group the input first before pointer chasing.

Another bit property set that has to be determined during the preparation step is the actual goal of the query. This can be implemented in multiple ways, the simplest solution is to create a "goal" rule that requires the query operations. Thus, no special code for minimization etc. is required; the requirements of this goal after the preparation step are the goals of the search phase. In our example the goal consists of *person.name/dep*, *department.name* and *map/join applied*.

5.7.2. Search Phase

Within the search space the plan generator tries to find the cheapest plan satisfying all bit properties required for the final solution. Note that we ignore performance optimization in the following discussion to make the conceptual structure clearer. Optimizations are handled separately in Section 5.7.4, and the algorithms are discussed in more detail in Section 6.3.2.

The core of the plan generator itself is surprisingly small and only consists of a single function that finds the cheapest plans with a given set of bit properties. The search phase is started by requesting the cheapest plan that provides the goal properties.

```
PLANGEN($goal$)
 1  $plans \leftarrow memoizationTable[goal]$
 2  if $plans$ is undefined
 3     then $plans \leftarrow$ create a new $PlanSet$
 4          $shared \leftarrow goal$ rewritten to use equivalence class representatives
 5          if $shared \cap goal = \emptyset$
 6             then $plans \leftarrow CALLplangen(shared)$
 7          for each $r$ in instantiated rules
 8             do if $r.filter \subseteq goal$
 9                   then $plans \leftarrow plans \cup \{r(p) | p \in$ PLANGEN$(goal \setminus r.produced)\}$
10          $memoizationTable[goal] \leftarrow plans$
11  return $plans$
```

What is happening here is that the plan generator is asked to produce plans with a given set of bit properties. First, it checks the memoization data structure (e.g. a hash table, bit properties→plan set) to see if this was already done before. If not it creates a new set (initially empty) and stores it in the

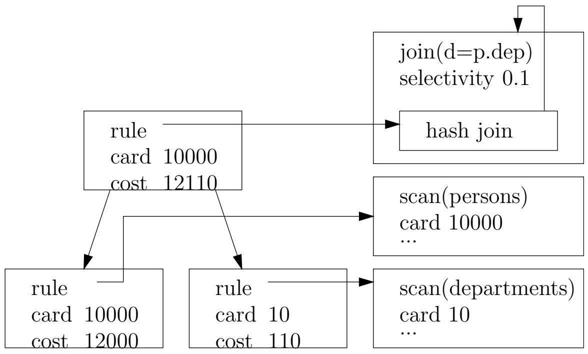

Figure 5.11.: A possible plan for the query in Figure 5.9

memoization structure. Then it checks if the goal can be rewritten to use only representatives from equivalence classes (if o_1 and o_2 are in the same equivalence class, the bit properties produced by o_2 can be replaced with the bit properties produced by o_1, see Section 5.6.3). If the rewrite is complete, i.e., the new goal is disjunct from the original goal, the current problem can be formulated as a new problem using only share equivalent operators. Thus, the plan generator tries to solve the new problem and adds the results to the set of usable plans. Afterwards, it looks at all rule instances, checks if the corresponding filter is a subset of the current goal (i.e. the rule is relevant) and generates new plans using this rule. Note that the lines 7-9 are very simplified and assume unary operators. In practice, the optimizer delegates the search space navigation (here a simple $goal \setminus r.produced$) to the rules. A more technical disucssion is given in Section 6.3.2.

As we will see in Section 5.7.4, a concrete implementation is more complex, as it tries to minimize search space, but the basic concept of the plan generator is both simple and elegant.

5.7.3. Reconstruction

The output of the search phase is a DAG of partial plans where every partial plan is annotated with a rule. This DAG has to be converted into an operator DAG with operators from the physical algebra. As the rules were instantiated specifically for the current query and, therefore, contain the required annotations, this conversion can be done in a single depth-first traversal. Shared partial plans must be detected during the traversal to create a DAG instead of a tree, but otherwise the transformation is straight-forward. This step constructs the final result of the plan generation that is used in the remaining phases of the compile time system.

For the example from Section 5.7.1, the final plan is shown in Figure 5.11: Note that the topmost plan node points to a rule that is hidden from the plan generator. The plan generator asked the join rule to produce a plan, and it decided to use a hash join; therefore, the *rule* member points to the hash join rule, which is embedded in the general join rule. During the reconstruction phase, this hash join rule is asked to produce an operator tree, which results in a simple hash join with two scans as inputs.

5.7.4. Optimizations

Pruning

While the algorithm described above is compact and produces the correct result, it examines too many plans. Consider the join ordering problem with cross products. The algorithm above will examine the full set of $2^n - 1$ logical partial plans (and even more physical plans), regardless of the relations or the joins involved. However, some of these plans will most likely be dominated by other plans, as they are simply too expensive. It makes sense to stop exploring the search space when it is obvious that the examined plans are more expensive than an already known solution. This is not only true for complete solutions but also for solutions for partial problems: If a plan for a given set of bit properties was already constructed, its cost can be used to prune plans which must satisfy the same bit properties. We will look into this in more detail below.

So this optimization consists of two steps: Determine a cost bound for searching and stop searching when the bound is violated. We will look at the local cost bounds for partial plans in the next paragraph, but it also makes sense to determine a global cost bound first before starting the search space: Thus, the plan generator can already prune partial plans, although no complete alternative for the current subproblem has been constructed. A very loose global bound can be found by simply considering the canonical execution plan; if a partial plan is more expensive than the canonical execution plan, it can be safely discarded. However, this bound does not prune many plans, as it is too high. A better bound can be found by using e.g. a greedy heuristic for join orderings or, even better, a more advanced heuristic like KBZ [43]. These heuristics quickly construct a plan that is much better than the canonical plan and which provides a tighter cost bound for the search space.

During the search phase the cost bound can be lowered for subproblems if it is known that a cheaper plan for the subproblem exists. However, one problem with maintaining a local bound is that plans are sometimes not directly comparable: For example, two plans might produce the same bit properties, but differ in the orderings they satisfy. The same can be true for other characteristics that are not encoded in the search space but only enforced on demand, like grouping and physical location. In order to derive a safe bound, the costs of a plan have to be increased by the maximum costs required to satisfy any of these characteristics. When considering only ordering and grouping, these are the costs for a sort, as this can guarantee any ordering and grouping. These optimizations are sketched for join generation below (the code is actually a function used by the rules and not part of the plan generator itself):

```
GENERATEJOINS($goal, bound$)
1  $result \leftarrow \emptyset$
2  $space \leftarrow goal \setminus produced$
3  for $\forall lp, rp : lp \cup rp = space \wedge lp \cap rp = \emptyset$
```

```
4     do for $\forall l \in$ PLANGEN$(lp, bound), r \in$ PLANGEN$(rp, bound)$
5         do $p \leftarrow$ join $l$ and $r$
6             if costs of $p \leq bound$
7               then $result \leftarrow result \cup \{p\}$
8                    $pc =$ costs of $p +$ costs to sort $p$
9                    $bound \leftarrow min(bound, pc)$
10  return $result$
```

The *bound* parameter is used to determine whether a plan is more expensive than an already known alternative. It is also passed to the recursive plangen steps and updated if a cheaper plan was found. Thus, the search space gets smaller whenever the plan generator can get a more accurate estimation of the costs.

When implemented naively, this cost-based pruning conflicts with memoization. The reason for this is that the plan generator can be asked multiple times, but with different bounds for plans with a given set of bit properties. If, for example, on the first call the cost bound was very low, the plan generator did not find a plan which is cheap enough. However, on a further call the bound might be higher, allowing for a plan which is cheap enough. As the memoization mechanism described above ignores the bounds, an empty set will be returned although a possible plan exists. The problem could be avoided by making sure that the bound never increases for a given set of bit properties. While this is theoretically possible using a suitable search space exploration strategy, it is difficult to do so: A problem can be part of multiple larger problems, which themselves can have very different cost characteristics [54]. Deciding beforehand which problem should be explored first is difficult.

Using the bounds together with the bit properties as key for the memoization is not an option, as the bounds will be different for nearly all calls. As the problem only occurs if no solution was found (otherwise, raising the bounds does not change the optimal plan), a simple solution would be to ignore the memoized result if the plan set is empty. This is not advisable, as it could trigger cascades of redundant searches with exponential runtime complexity. A better solution would be to store the bounds together with the plan set, so that the plan generator can check if the current bounds are actually higher and only searches in this case.

Apart from storing the bound used for searching in the plan set, it is also beneficial to store a lower cost bound for the plan set: Each plan set should store the minimum costs of a plan that would satisfy the required bit properties of the plan set. Initially, this lower bound could be set to zero, but during the optimization phase the bound can be raised: Consider a situation where no plan for a subproblem can be found, as all alternatives are more expensive than the current bound. However, some plans will have been considered, either complete plans (for this problem) or at least partial plans for subproblems. These plans were discarded, as they were too expensive, but their costs can be used as lower bounds; a plan for the whole problem will cause at least the costs of the cheapest alternative that was discarded. The great advantage of a lower bound is that the plan generator can now decide if the subproblem should be explored again when the bound has been raised: If the lower bound is higher than the current upper bound, the subproblem can be ignored.

Ordering Predicates

For some operators, especially selection predicates that do not share common subexpressions, an optimal execution sequence can be computed before the search phase. For a more detailed discussion see [59], but in principle the optimal execution sequence can be computed using the formula

$$rank = \frac{1 - selectivity}{costs}.$$

The plan generator does not use this information, as is tries each operator independently of the other operators. But exploiting the rank can be added quite easily:

Although for all rules described so far each rule corresponds to exactly one physical operator in the generated plan, this is not necessarily the case: When a rule for a predicate is asked to create plans containing this predicate, it can check which other predicates are also required and create plans containing these predicates in optimal ordering. In this case the operator rule handles more than one operator, but this does not interfere with the plan generation itself, as the plan generator expects the rules to navigate the search space and assumes nothing about the number of operators involved.

While this ordering by rank is the simplest to implement, more advanced algorithms that also consider common subexpressions (see [59]) could also be embedded in the operator rules. This would only require a local change in the rule for selection operators.

Early Plans

An unfortunate property of construction-based plan generators is that they do not produce incrementally better solutions but just the optimal solution after the whole algorithm is finished. This is extreme for bottom-up plan generators, as they start with small plans first and only produce nearly complete plans when the algorithm is almost done. As the optimization time for complex queries can be quite large, this is unfortunate, as the user might prefer a suboptimal execution plan over waiting for the plan generator. Ideally, the plan generator should produce suboptimal but complete plans from time to time to allow aborting the search phase earlier.

For transformation-based plan generators this is trivial, but it can also be done for constructive top-down plan generators: The search phase consists of recursive calls to operator rules which themselves find the optimal solution for a subproblem. When the plan generator decides it needs an early plan, the operator rules report the best plan found so far to their callers, which can either use it or use a better plan already found one level higher, until a complete plan reaches the root of the search process [39].

Problematic are operators that require two input plans, for example joins. Although the join rules try different left/right combinations successively, for a huge search space it might happen that only one side has been examined. Then no partial plans for the other side have been examined and, therefore, no complete plan can be constructed. A solution for this problem is to combine the operators of the missing side using a simple heuristic, which will produce a suboptimal but at least complete plan without spending a lot of time searching.

Besides offering fast answers to the user, these early plans have the additional advantage of reducing the search space. As the search phase proceeds, the early plans will get better and, thereby, reduce the global cost bound. The only disadvantage is the overhead to construct these early plans, but if they are constructed every hundred thousands partial plans or so the overhead will be negligible.

5.7.5. Top-Down vs. Bottom-Up

The description of the plan generator so far assumed that the plans are constructed top-down. This means that the algorithm starts with the whole problem (all bit properties required for the query) and splits it into smaller parts that are solved recursively. Another approach is to construct the plans bottom-up. There the plan generator starts with the smallest possible plans (e.g. table scans) and combines these plans using operators until the whole query has been constructed. Both approaches find the same solution and for many problems (e.g. only joins and selections) also generate the same intermediate plans. In fact, most rules can be transformed into bottom-up rules easily:

```
SELECT::SEARCH($plans, goal$)
1  for each $p \in$ PLANGEN($goal \setminus produced$)
2     do $p' \leftarrow$ add the selection to $p$
3        $plans \leftarrow plans \cup \{p'\}$
  $< - >$
SELECT::SEARCHBOTTOMUP($plans, current$)
1  for each $p \in plans$
2     do $p' \leftarrow$ add the selection to $p$
3        $dpTable[current \cup produced] \leftarrow dpTable[current \cup produced] \cup \{p'\}$
```

Instead of removing the bit properties provided by the rule to identify the subproblem, the rules get the subproblem and add the bit properties provided by them. For joins the rules determine which side they get (left or right), determine the bit properties of the other side, look up the corresponding plans in the dynamic programming table, construct a new plan and store it in the table at the correct position.

While the two algorithms find the same solution and have the same time complexity, they behave somewhat differently. The top-down approach has three advantages: First, it is more intuitive to write rules this way (similar to a top-down vs. a bottom-up parser). Second, after a while the plan generator already knows solutions for relatively large subproblems. This allows a much better cost bound propagation. Experimental results showed a search space reduction by $10 - 20\%$. The construction of early plans is related to this, as discussed above. Third, the top-down approach only considers subproblems if they make sense later on, while the bottom-up approach tries any combination of operators. However, this only makes a difference if operators are freely combinable, e.g., when considering two plan alternatives with disjunct operator sets. The great advantage of the bottom-up method is that it only considers operator combinations that are actually possible, as it constructs them this way. Often the top down approach tries to solve subproblems for which no solution exists.

This is a real problem. For chain queries with 10 relations $> 99.9\%$ of all constructed plan sets are empty when ignoring cross products. These empty plan sets not only waste memory, but also consume a lot of time searching for a result, which means that the top-down approach is much slower than the bottom-up approach. While this is only true when not considering cross products, eliminating the problem by greatly increasing the search space is no option. The severity of the problem can be reduced by inserting a sanity check (lines 3-5) into the plan generator: Before solving a subproblem, it checks if there exists an operator combination whose combined bit properties produce the desired goal.

```
PLANGEN($goal$)
 1  $plans \leftarrow memoizationTable[goal]$
 2  if $plans$ is undefined
 3    then $mask \leftarrow \bigcup_{r \in \text{ instantiated rules } : r.filter \subseteq goal} r.filter$
 4         if $mask \neq goal$
 5           then return $\emptyset$
 6         $plans \leftarrow$ create a new $PlanSet$
 7         $shared \leftarrow goal$ rewritten to use equivalence class representatives
 8         if $shared \cap goal = \emptyset$
 9           then $plans \leftarrow plans \cup$ PLANGEN($shared$)
10         for each $r$ in instantiated rules
11           do if $r.filter \subseteq goal$
12                then $plans \leftarrow plans \cup \{r \circ p | p \in \text{PLANGEN}(goal \setminus r.produced)\}$
13         $memoizationTable[goal] \leftarrow plans$
14  return $plans$
```

This eliminates a lot of unnecessary searches and, in fact, reduces the runtime by more than a factor of 20 for large queries. But the check itself requires a time linear in the number of operators. For large queries this means that $> 90\%$ of the total CPU time is spent on this check. The plan generation is still much faster than without the check, but as a consequence, the bottom-up approach, which does not have this problem, becomes faster for large problems. This is shown in Figure 5.12. For small problems, the two approaches are about the same, the top-down approach is somewhat faster due to better pruning. But for larger queries, the linear costs during a top-down search are noticeable and the heuristic does not eliminate all fruitless tries either. A related problem occurs when a rule constructs a DAG: As the subproblems overlap, the rule has to perform many identical calls to the plan generator. While this does not increase the search space (due to memoization), it involves many fruitless table lookups. A bottom-up approach is more efficient here, as it knows which plans

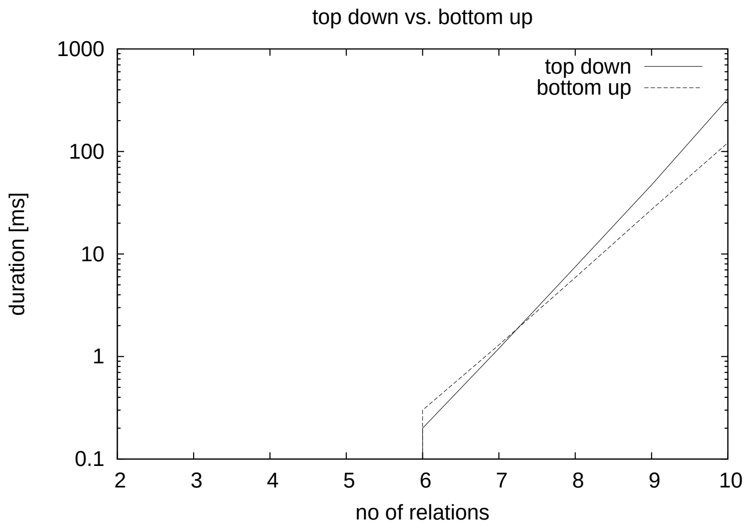

Figure 5.12.: Plan generation for chain queries, average of 100 runs

are available. Perhaps a faster implementation of the search space pruning check is possible (e.g. using a decision tree of applicable operators), but as performance was not the main goal of this work, we just continue to use the described top-down approach, as it can be reformulated as a bottom-up search easily.

5.8. Example Plans

To illustrate the different aspects of plan generation, we now consider the whole process for a more complex query. Consider the OQL query shown in Figure 5.13. It first computes the revenue for each project (creating the temporary view `prjsum`) and then selects all persons that work in Germany and lists them together with their project that created the highest revenue. A possible logical representation is shown in Figure 5.14. Note that this is not meant to imply an execution plan! While the final execution plan is similar in this example, the figure is just an illustration. The query compiler considers very different plans. We assume that unnesting and rewriting has happened before the plan generation phase. The query is answered by selecting all persons who work in Germany (a sequence of pointer chasing χ operations), determining the projects of them (the μ operator unnests the set-valued attribute `projects`) and joining the result with the view (which just consists of a join and a group-by, see the appendix for a definition of Γ). Now the intermediate result is used twice. One branch performs a group-by to determine the maximum revenue for each person and joins the result to the other branch to get the projects with the highest revenue. Note that this could not be done with an aggregation, as multiple projects could have the highest revenue. Finally, the result is sorted on the attribute m.

```
define prjsum(project,sum) as
   select p, sum(
      select o.total
      from   orders o
      where  o.project=p)
   from projects p

select   p,pr, s.sum
from     persons p, p.projects pr, prjsum s
where    p.group.department.country = "D" and
         pr = s.project and
         s.sum=max(
            select t.sum
            from   prjsum t
            where  t.id in p.projects)
order by s.sum desc
```

Figure 5.13.: More complex OQL query

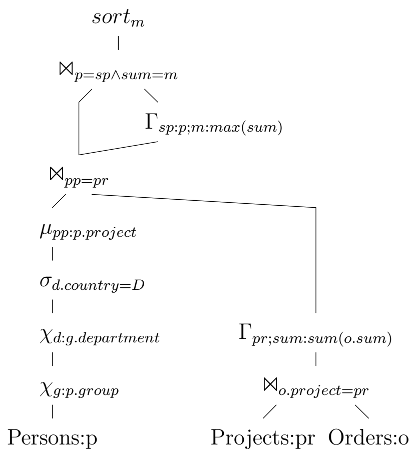

Figure 5.14.: Logical algebra representation of Figure 5.13

In the preparation step, the plan generator examines this representation and instantiates operator rules as needed. The resulting instances are shown in Figure 5.15. The rules $1-12$ can be derived directly from the query representation, rules $13-16$ correspond to a transformation of pointer chasing into joins, and rules $17-18$ perform the same lookup by using an access support relation *persons.group.department.country* instead (ASR, [38]). The sort operator does not need a rule, as orderings are handled separately. Note that we omitted some rules here to make the example more readable: All rules except 11 and 12 actually occur twice, once on the left-hand side of the final join and once below the group-by (this is not strictly required, but typically the query is considered a tree by the previous steps until it is converted into a DAG by the plan generator). We only show one version here, as in fact the plan generator uses just one representative in the result, but rule 11 requires p' and sum' instead of p and sum to make it clear that it requires a different version (handled by renames). Now the set of bit properties is minimized. Many bit properties can be eliminated, as they are never tested for (e.g. o_1, o_2, o_3). Others are merged, as they are only produced in combination (e.g. sp, m and o_9, sum). The minimum goal of the query specified by the user is p, pr, sum, o_6, o_{12}, which means that the projected attributes must be there and all selections were applied (one selection was transformed into a join).

However, we also want to partition the search space by applied operators (to avoid endless loops and to reduce fruitless tries). Therefore, we increase the goal of the query. For example, the bit property pp is required by one operator, and already while searching we want to distinguish between plans that will include it and plans that will not. Otherwise, operators that require pp would be scheduled, although the bit property cannot be produced in the current subproblem. So we specify all bit properties produced by rules $1-12$ (the logical representation) as goal and use the normal minimization mechanism (see Section 6.3.1) to reduce it. This results in the goal $p, pr, o, o_4, g, o_5, d, o_6, pp, o_8, sum, o_{10}, m, o_{12}$. Note that the concrete goal is somewhat arbitrary, as bit properties could have been merged differently. For example, we could use o_9 instead of sum, but preferred the attribute name due to readability.

Now the plan generator recursively starts generating plans. It selects some rules, breaks the global goal into smaller ones and solves these subproblems recursively. For example, it might choose the rule $\bowtie_{pp=pr}$ first, as it satisfies a goal of the query (o_{10}). However, it is not possible to construct the whole query using it as the topmost operator, so the try will fail. This is a disadvantage of the top-down approach, see Section 5.7.5 for a detailed discussion how to avoid it. The rule $\bowtie_{p=sp \wedge sum=m}$ can be used as topmost rule (but this is only known ex post, all rules are tried). It triggers recursive searches, with and without the group-by operator (bit property m). Although this seems to increase the search space a lot (as the same problem is solved multiple times), the memoization mechanism avoids this, the $\Gamma_{sp:p,m:max(sum)}$ rule strips the sp and the bit properties are identical afterwards (after a rename), resulting in a DAG.

Especially interesting is the solution of the subgoal p, o_4, g, o_5, d, o_6. It can either be solved by using the rules $1, 5-6$, by using the rules $1, 6, 13-16$, or by using the rules $1, 6, 17-18$. The different alternatives are shown in Figure 5.16:

id	type	requires	produces
1	Persons:p		o_1, p
2	Projects:pr		o_2, pr
3	Orders:o		o_3, o
4	$\chi_{g:p.group}$	p	o_4, g
5	$\chi_{d:g.department}$	g	o_5, d
6	$\sigma_{d.country=D}$	d	o_6
7	$\mu_{pp:p.project}$	p	o_7, pp
8	$\bowtie_{o.project=pr}$	o, pr	o_8
9	$\Gamma_{pr;sum:sum(o.sum)}$	pr, o	o_9, sum
10	$\bowtie_{pp=pr}$	pp, pr, o_9	o_{10}
11	$\Gamma_{sp:p,m:max(sum)}$	p', sum'	o_{11}, sp, m
12	$\bowtie_{p=sp \wedge sum=m}$	p, sp, sum, m	o_{12}
13	Groups:g		o_{13}, g
14	Departments:d		o_{14}, d
15	$\bowtie_{p.group=g}$	p, g	o_{15}, o_4
16	$\bowtie_{g.department=d}$	g, d	o_{16}, o_5
17	ASR:a		o_{17}, a, d
18	$\bowtie_{p=a.p}$	p, a	o_{18}, o_4, o_5

Figure 5.15.: Rule instances for Figure 5.14

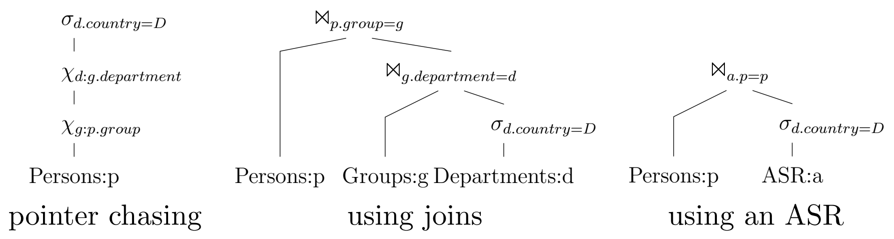

Figure 5.16.: Plan alternatives

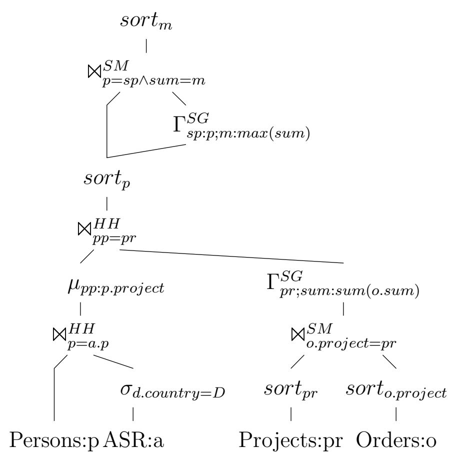

Figure 5.17.: Execution plan for Figure 5.13

The first alternative uses pointer chasing as stated in the query, the second replaces this with joins (other join orderings and join types were omitted), and the third alternative uses an access support relation to calculate the whole path in one step. As all alternatives can satisfy the requested bit properties, the plan generator is free to choose.

The final result of the plan generation step is shown in Figure 5.17. While this strongly resembles the input for this specific query, the plan generator chooses to use the ASR instead of pointer chasing and replaced the logical joins and group-bys with concrete join and group-by implementations. It also added sort operators as needed. Note that the join/group-by on the right-hand side could be done more efficiently if a hash-based join operator guaranteed output clustered on the group-by attribute (see e.g. [26]), but we did not assume such an implementation here.

5.9. Evaluation

When modifying a plan generator to support DAGs, it is essential to check how this change influences the runtime behavior. To study the effect of the query graph structure, we started with a chain query with 10 relations and then increased the number of equivalent relations: first the query $A \bowtie B \bowtie C \ldots J$, then $A \bowtie A \bowtie B \ldots I$ etc. The runtime of the plan generator is shown in Figure 5.18: Until the number of shared relations is about 5, the runtime shrinks drastically, as the number of considered alternatives is reduced (the shared relations are equivalent to each other). For more than 6 relations, the runtime increases again, as the join graph becomes more and more a clique. For 9 shared relations (one relation is read 10 times), the join graph is a clique and the search space is as large as if the plan generator would consider cross products. This effect aside, recognizing equivalent expressions actually decreases the search space compared to a purely tree-based optimizer.

Note that this is not an objective way to measure the costs for DAG support,

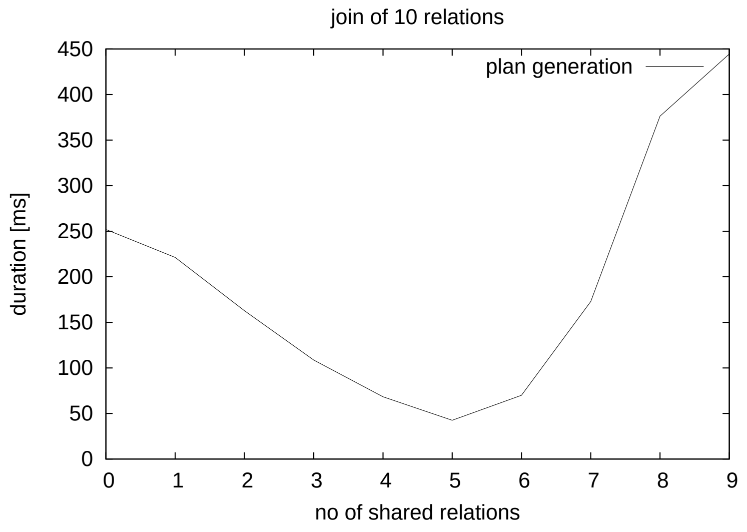

Figure 5.18.: Join of 10 relations with an increasing amount of sharing

as by increasing the amount of equivalent relations we also change the semantics of the query. A fairer comparison is done in Chapter 11, here we are only interested in how the plan generation is affected by the query.

The good performance of the DAG-creating plan generator is based upon the forced sharing approach, which means that all share equivalent plans are pruned against each other and only the cheapest is kept. If sharing is optional, i.e. both sharing and not sharing plans are considered at the same time, the search space is increased considerably. This is shown in Figure 5.19: When making sharing optional, the runtime increases instead of decreasing when the number of shared relations increases. Note that this is even a situation where the top-down approach performs well, as it only considers the relevant sharing alternatives. When using a bottom-up approach, the forced sharing is faster than the top-down approach, but with optional sharing the search space explodes, as every possible sharing variant is constructed. So for the plan generator, forced sharing is highly recommended when constructing plans to keep the search space reasonable.

5.10. Conclusion

The plan generator described here allows a very flexible generation of DAG-structured query graphs. It is independent of a concrete database system and can support a wide range of operators. Creating DAG-structured query graphs is not much more difficult than creating trees and, in fact, reduces the search space when sharing is possible. Therefore, it makes sense to create DAGs even though the runtime system does not support it: Plan generation is faster when sharing nodes, and after plan generation the plans can be converted back into trees or temp operators can be added as needed.

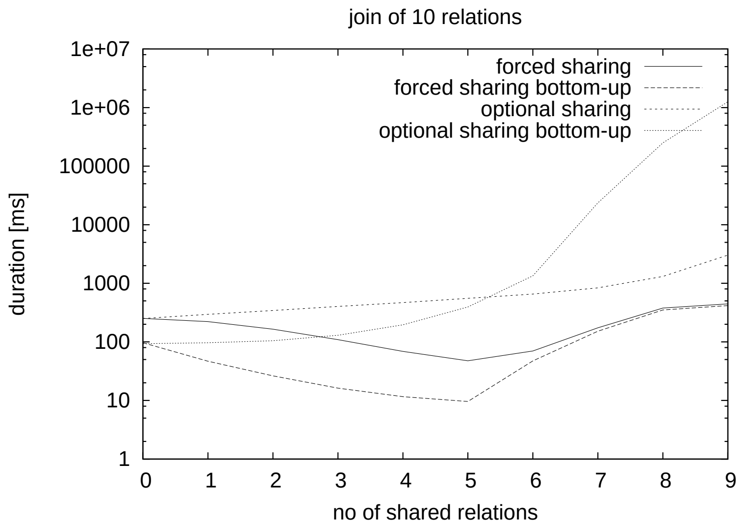

Figure 5.19.: Join from Figure 5.18 with optional sharing

Further work should examine how to degrade gracefully if the search space is too large for memoization or dynamic programming. For tree-structured query graphs techniques have been proposed [40], but this becomes more complicated when taking sharing into account, as alternatives may be interesting only because of result reusage. A related topic is taking the query structure into account: In some scenarios (e.g. TPC-H query 2, see Section 11.2) the query can be partitioned into parts that can be optimized (nearly) independently of each other, which reduces the search space very much. However, this is more difficult to do when generating DAGs, as sharing has to be taken into account across different plan generator runs.

6. Algorithms for DAG Generation

After giving a high-level view of generating DAG-structured query plans in Chapter 5, we now go into more detail about the algorithms and data structures. Note that this chapter deals more with technical details, the general discussion has been done in the previous chapter.

6.1. Data Structures

Memoization Table

The plan generator uses memoization to reuse already solved subproblems. Therefore, the search space is partitioned by the bit properties, and the memoization table is a hash table using the bit properties as key. As multiple uncomparable plans can satisfy the same bit properties (e.g. because of different orderings), the memoization table contains a *PlanSet* entry instead of a single plan for each examined bit property combination:

$$memoTable : bitset \rightarrow PlanSet$$

Plan Sets

As multiple plans can satisfy the same bit properties, they are organized in sets that maintain the common data:

PlanSet

properties	: bit set
state	: logical state for the cost model
plans	: set of *Plan*

The attribute *properties* is a bit set containing all bit properties satisfied by the plans in the plan set. It is stored here instead of the plans themselves in order to conserve space. The attribute *state* contains the logical state shared by the plans, which is relevant for the cost model. The state is discussed in Section 9.3.4 and usually includes the cardinality, the average size of each tuple, etc. For the plan generator itself, this state is not significant, it is only used and maintained by the cost model. Finally, the attribute *plans* is a set of all plans with the stated bit properties.

Plans

The plans themselves keep as little information as possible; mainly the costs, the rule that created the plan and the subplans:

Plan		
rule		: *Rule* instance
physicalProperties		
	ordering	: ordering/grouping state
	costs	: costs of the cost model
	shared	: bit set
left		: *Plan* (for unary/binary rules)
right		: *Plan* (for binary rules)

The attribute *rule* specifies the rule that created the plan node. This is used in the reconstruction phase to create the actual operator tree. Usually, the *rule* entry corresponds to one physical operator (e.g. for joins the generic join rule selects a join implementation and sets *rule* to point to a rule describing the implementation). The physical properties are stored in *physicalProperties*: The attribute *orderings* contains the orderings and groupings properties of the current plan node, it is just a pointer inside the ordering and grouping framework described in Chapter 8. The attribute *costs* consists of the cost description of the cost model. As with the *state* entry, this attribute is opaque to the plan generator. The attribute *shared* specifies the (potentially) shared operators in this plan (see Section 5.6.3). It is used by the cost model to detect DAG generation and to decide whether plans are comparable. Finally, the attributes *left* and *right* are used to model the actual operator tree (respective DAG), for unary operators only one entry is used and operators that require no input use neither of them.

It is noticeable that the plan generator ignores most of the attributes (all but *rule* and *properties* in *PlanSet*). This is due to the rule-based structure of the plan generator. The actual cost computations are all done inside the rules referenced by *rule*, the plan generator itself only cares about required and provided bit properties. This results in a very clean and compact plan generator that is also very extensible.

6.2. Rule Interface

Before looking at the concrete algorithms, we first consider the coupling between the plan generator and the rules. The rules are only visible as abstract data types (ADT) and offer the required functionality as methods. At the interface level, there is a distinction between three types of rules, generic rules and two specializations. The generic rule interface is shown below:

Rule	
UPDATEPLAN(*Plan*)	: *void*
BUILDALGEBRA(*Plan*)	: *PhysicalAlgebra*

The method UPDATEPLAN describes the semantics of the query: Given a plan with *left* and *right* members set to suitable subplans, update the remaining members (costs, cardinality, ordering etc.) according to the rule semantics. This is useful when the structure of a subplan is known and only the details have to be filled in (e.g. the plan generator inserted an explicit sort). The second method BUILDALGEBRA takes a complete plan and converts it into the physical algebra. This is only used during the reconstruction phase to construct the final result.

Rules that only offer the *Rule* interface are mainly helper rules like sort or temp. More important for the plan generator are rules that can influence the exploration of the search space. The interface is shown below, note that the ADT is a specialization of *Rule*.

SearchRule : *Rule*
  filter : bit set
  SEARCH(*PlanSet*) : *void*

The important method is SEARCH. It takes a *PlanSet* structure and tries to construct plans with the bit properties specified by the *PlanSet*. We will see an example of this in Section 6.3.2. The attribute *filter* is used for performance optimization: It contains all bit properties produced and required by the rule. Thus, the plan generator can decide quickly if the rule is relevant for the current subproblem (i.e. *filter* is a subset of currently explored bit properties) and calls SEARCH only in this case.

The second specialization is also a performance optimization, it could be replaced with *SearchRule*. It describes rules that require no input:

BaseRule : *Rule*
  INITIALIZE(*PlanSet*) : *void*

Note that *BaseRule* is derived from *Rule* and not from *SearchRule*, although it explores the search space. This is due to the fact that the plan generator knows that the descendants of *BaseRule* require no input: As they can produce exactly one set of plans, these plans are produced and stored as partial results in the memoization table before the actual search. Otherwise, the plan generator would always consider using this rule, which would succeed only once. By using a separate class, the plan generator can handle table scans etc. much more efficiently. The INITIALIZE method sets the attributes of the plan set to the characteristics of the rule, i.e., sets properties, cardinality, and tuple size. This is used only once before the search phase when memoizing the initial plans.

6.3. Plan Generation

6.3.1. Preparation Phase

The preparation phase examines the query and collects the information necessary for plan generation. This was already discussed in Section 5.7.1, here we examine two more complex steps, namely the property minimization and the construction of equivalence classes for share equivalence.

Properties

To make the plan representation more compact (and also to reduce the search space), the preparation phase prunes and minimizes the property specification. This consists of three steps: First, all bit properties are collected. Thereby, produced and required bit properties are kept separate, resulting in two sets of bit sets. The algorithm keeps each individual bit property combination (and, therefore, produces a set of bit sets instead of a bit set), as this is required to check which bit property combination can be produced.

```
COLLECTPROPERTIES()
1  $produced \leftarrow \emptyset$
2  $required \leftarrow \emptyset$
3  for each $r$ in instantiated rules
4     do $produced \leftarrow produced \cup \{r.produced\}$
5        $required \leftarrow required \cup \{r.required\}$
6  return $(produced, required)$
```

Besides the property specifications, the preparation phase also determines the goal of the query (i.e. the bit properties the final plan must satisfy). This could be done in multiple ways. One possibility is to collect all bit properties of the logical operators in the original query (see Section 7.2).

```
BUILDGOAL()
1  $goal \leftarrow \emptyset$
2  for each $r$ in instantiated rules
3     do if $r$ represents a logical operator in the query
4           then $goal \leftarrow goal \cup r.produced$
5  return $goal$
```

Now this information can be used to prune properties. The algorithm checks which bit properties can be used to construct the *goal* properties (potentially transitively) and removes all bit properties that are not useful. Afterwards, it checks which bit properties can be satisfied by the remaining produced properties and eliminates all that cannot be satisfied. Note that while the algorithm shown below only minimizes the property sets, the preparation phase afterwards removes all rules whose bit properties are no longer relevant or cannot be satisfied. The parameters *produced*, *required* and *goal* are the result of COLLECTPROPERTIES and BUILDGOAL.

```
PRUNEPROPERTIES($produced, required, goal$)
1  repeat
2        $useful \leftarrow goal$
3        for each $r$ in instantiated rules
4           do if $r.required \in required \wedge r.produced \cap useful \neq \emptyset$
5                 then $useful \leftarrow useful \cup r.required$
6        for each $r$ in instantiated rules
7           do if $r.produced \cap useful = \emptyset$
8                 then $produced \leftarrow produced \setminus \{r.produced\}$
9        $possible \leftarrow \emptyset$
```

```
10             for each $r$ in instantiated rules
11               do if $r.produced \in produced \wedge r.required \subseteq possible$
12                    then $possible \leftarrow possible \cup r.produced$
13             for each $r$ in instantiated rules
14               do if $r.required \not\subseteq possible$
15                    then $required \leftarrow required \setminus \{r.required\}$
16      until $produced$ and $required$ are not modified in this pass
17   return $(produced, required)$
```

The remaining bit properties are now minimized. While all remaining bit properties can be produced, they can still contain irrelevant entries (if other entries in the same bit set are relevant). These are removed, and all bit properties that are always produced together are merged.

```
MINIMIZEPROPERTIES($produced, required$)
 1   $result \leftarrow \emptyset$
 2   $R \leftarrow \bigcup_{r \in required} r$
 3   for each $p$ in $produced$
 4      do for each $b$ in $p$
 5            do if $b \notin R$
 6                  then $p' \leftarrow p \setminus \{b\}$
 7                       $produced \leftarrow (produced \setminus \{p\}) \cup \{p'\}$
 8                       $p \leftarrow p'$
 9                       $result \leftarrow result \cup \{b \rightarrow \emptyset\}$
10   $P \leftarrow \bigcup_{p \in produced} p$
11   for each $a$ in $P$
12      do for each $b$ in $P, a \neq b$
13            do $A \leftarrow \{p | p \in produced \wedge a \in p\}$
14               $B \leftarrow \{p | p \in produced \wedge b \in p\}$
15               if $A = B$
16                  then $result \leftarrow result \cup \{ab \rightarrow a\}$
17   return $result$
```

The produced map function is then used to adjust the bit properties of the remaining rules. When calculating the map function as shown, the mapping is ambiguous (when two bit properties can be merged, any of the two can be chosen). In practice, this is solved by defining an arbitrary total ordering among bit properties and changing $a \neq b$ in line 12 into $a < b$.

Share Equivalence

After determining which rules are relevant, the preparation phase decides which rules are share equivalent. So far, we have only given a formal definition of share equivalence for algebraic expressions, but the plan generator must also know which operators could be used to build share equivalent plans. We consider two operators as share equivalent if they form share equivalent expressions given share equivalent input. For example:

$$\bowtie_1 \equiv_S \bowtie_2 :\prec\succ \forall_{A_1,A_2,B_1,B_2} A_1 \equiv_S A_2 \wedge B_1 \equiv_S B_2 \Rightarrow (A_1 \bowtie_1 B_1) \equiv_S (A_2 \bowtie_2 B_2)$$

The plan generator uses this notion for rules (which can consist of multiple operators), but in practice, this is no problem: The rules are treated as if they were (complex) operators. In general, the plan generator tests if two rules are structurally identical (i.e. belong to the same class, have predicates with the same structure etc.). Then, if a mapping can be found such that one rule can be replaced by the other rule and a rename, they are considered share equivalences:

```
SHAREEQUIVALENTRULES($r_1, r_2$)
1  if $r_1$ and $r_2$ are structural identical
2     then $P_1 \leftarrow \{r | r \in$ rules $\wedge\, r.produced \subseteq r_1.required\}$
3          $P_2 \leftarrow \{r | r \in$ rules $\wedge\, r.produced \subseteq r_2.required\}$
4          if $\forall i \in P_1 \exists j \in P_2$ : SHAREEQUIVALENTRULES($i, j$)
5             then if $\forall i \in P_2 \exists j \in P_1$ : SHAREEQUIVALENTRULES($i, j$)
6                     then return $true$
7  return $false$
```

The algorithm builds the mapping implicitly, by recursively testing if the input of the two operators is share equivalent. It might even be easier to construct the mapping explicitly in a bottom-up fashion, starting with share equivalent scans and then transitively considering their consumers. The equivalence relation is now used to construct equivalence classes, selecting one rule as representative and discarding equivalence classes with only one element (as no sharing is possible then).

```
CONSTRUCTEQUIVALENCCLASSES()
1  $C \leftarrow \emptyset$
2  for each $r$ in instantiated rules
3     do if $\exists (a, b) \in C$ : SHAREEQUIVALENTRULES($a, r$)
4           then $C \leftarrow C \setminus \{(a, b)\} \cup \{(a, b \cup \{r\})\}$
5           else $C \leftarrow C \cup \{(r, \{r\})\}$
6  for each $(a, b)$ in $C$
7     do if $|b| = 1$
8           then $C \leftarrow C \setminus \{(a, b)\}$
9  return $C$
```

These equivalence classes are used for two purposes: First, each representative is assigned a bit in the *sharing* property of the plans. Second, for all other entries in an equivalence class a mapping from their produced properties to the properties produced by the representative is constructed. Thus, the plan generator can rewrite a bit property set in terms of equivalence class representatives (by applying all applicable mappings) to try using a shared plan.

6.3.2. Search Phase

After the preparation phase, the search is started top-down with the minimized *goal* of the query; the pseudo-code is shown in Figure 6.1. In each step the plan

```
PLANGEN($goal$)
 1  $plans \leftarrow memoizationTable[goal]$
 2  if $plans$ is undefined
 3    then $mask \leftarrow \emptyset$
 4         for each $s$ in rules
 5         do if $s.produced \subseteq goal$
 6              then $mask \leftarrow mask \cup s.produced$
 7         if $mask \neq goal$
 8           then return $\emptyset$
 9         $plans \leftarrow$ create a new $PlanSet$ with $properties = goal$
10         $shared \leftarrow goal$ rewritten to use equivalence class representatives
11         if $shared \cap goal = \emptyset$
12           then $plans \leftarrow plans \cup$ PLANGEN($shared$)
13         for each $s$ in search rules
14         do if $s.filter \subseteq goal$
15              then $s$.SEARCH($plans, goal$)
16         $memoizationTable[goal] \leftarrow plans$
17  return $plans$
```

Figure 6.1.: Algorithm for plan generation

generator first checks the memoization table whether the problem was already solved. If not, it checks if the bit property combination could be produced by any rule combination, and stops the search if not (lines 3-8, see Section 5.7.5). To share equivalent plans, the plan generator now uses the mapping from operators to their equivalence class representatives constructed in the preparation phase to rewrite *goal* in terms of equivalences class representatives (lines 10-12). If this is completely possible (the new goal is disjoint from the old one), the plan generator solves the new goal and uses the result as the result for the current problem. The check is $shared \cap goal = \emptyset$ instead of $shared \neq goal$, as only whole subproblems can be shared. If *shared* and *goal* overlap, at least one operator remains that has to be scheduled first before sharing is possible (so sharing will be tried later during the recursive search).

Afterwards, the plan generator looks at all known instances of *SearchRule*, checks if their *filter* is a subset of *goal* (if not, the rule cannot produce a plan with the desired bit properties). If yes, it asks the rule to build a plan and to store it in the plan set. The plan set is passed down instead of the rule returning a plan, as there can be more than one plan for a given goal and, besides, the rules can use already known plans for pruning.

The rules direct the navigation of the search space. We give some illustrating examples here, a more thorough discussion can be found in Chapter 7. Consider a simple selection operator. When the plan generator asks a selection rule to generate a plan, the selection asks the plan generator to produce a plan without the selection and then adds the selection (note that *plans* is an instance of *PlanSet*, which automatically prunes dominated plans):

```
SELECT::SEARCH($plans, goal$)
1  for each $p \in$ PLANGEN($goal \setminus produced$)
2     do $p' \leftarrow$ add the selection to $p$
3        $plans \leftarrow plans \cup \{p'\}$
```

For binary operators like joins, the navigation is more complex. However, most of the required functionality is identical for all binary operators and, therefore, can be factored in common rule fragments in an implementation. The general rule for joins is shown below:

```
JOIN::SEARCH($plans, goal$)
1  $space \leftarrow goal \setminus (produced \cup requiredLeft \cup requiredRight)$
2  for each $lp \subseteq space$
3     do $rp \leftarrow space \setminus lp$
4        for each $l$ in PLANGEN($lp \cup requiredLeft$)
5           do for each $r$ in PLANGEN($rp \cup requiredRight$)
6                 do $p \leftarrow$ join $l$ and $r$
7                    $plans \leftarrow plans \cup \{p\}$
```

The joins first check which bit properties can be satisfied arbitrarily, i.e., are neither produced nor required by itself. These are called *space* here, as they actually describe the search space for the join rule. The rule has to decide which of these bit properties must be satisfied by the left subplan and which by the right subplan. This is done by enumerating all subsets of *space* and asking the plan generator for plans satisfying these requirements (and the requirements of the join itself). These plans are then combined to a new partial plan and memoized if they are cheaper than the existing plans.

Consider, for example, a problem with two relations R_1 and R_2, a join $\bowtie_1$ between them and a selection σ_1 (for simplicity reasons, we identify the bit properties with the operators here). Now the join rule is asked to produce a plan with the bit properties $R_1, R_2, \bowtie_1, \sigma_1$. The join itself produces the property $\bowtie_1$ and it requires R_1 and R_2. The only remaining bit property is σ_1, so $space = \sigma_1$. The join rule does not know on which relation the selection can be applied, so it first asks the plan generator to produce plans with R_1 and σ_1, and then asks for plans with R_2, which are joined. Afterwards, it asks for plans with R_1 and plans with R_2 and σ_1, which are also joined. As the selection can probably only be applied to one relation, one of these sets will be empty, but the join rule does not understand the semantics of the selection and, therefore, tries both possibilities. Note that this does not imply that selections are pushed down: The selection rule itself could have been scheduled before, resulting in $space = \emptyset$. In this case, the join rule would only consider a single plan.

6.3.3. Reconstruction

The reconstruction phase is mostly straight-forward, as the plan generator simply calls BUILDALGEBRA in the root of the final query plan to get the physical algebra. The only problem is that some additional information is required: First, the plans have no reference to their enclosing plan set, so all information stored there (especially the bit properties) is unavailable. Second, the graph forms a DAG, which means that plan nodes are visited multiple times, although they correspond to only one physical operator. Third, the renames due to share equivalence are only implicit during plan generation and have to be converted into explicit renames. Still, this can be done easily by storing the required information in a hash table that is used during reconstruction. The hash table is a map plan → (algebra, bit properties) and is filled during a depth-first search. For a selection, the reconstruction code is sketched below:

```
SELECT::BUILDALGEBRA($plan$)
 1  $(algebra, properties) \leftarrow reconstructionTable[plan]$
 2  if $(algebra, properties)$ is undefined
 3     then $input \leftarrow plan.left.rule.$BUILDALGEBRA$(plan.left)$
 4          $ip \leftarrow reconstructionTable[plan.left].properties$
 5          if $required \not\subseteq ip$
 6             then
 7                  $nip \leftarrow$ rename $ip$ by checking the operators in $plan.left$
 8                  $input \leftarrow$ add new rename operator $ip \rightarrow nip$ to $input$
 9                  $ip \leftarrow nip$
10          $algebra \leftarrow$ add new select operator to $input$
11          $properties \leftarrow ip \cup produced$
12          $reconstructionTable[plan] \leftarrow (algebra, properties)$
13  return $algebra$
```

The rule first checks if the physical algebra expression has already been constructed. If not, it requests the physical algebra expression of its input and looks up the corresponding properties. If the requirements of the rule cannot be satisfied by these properties, a rename is required. The subplan is scanned to find this rename (the plan must contain equivalence class representatives who are equivalent to rules that can produce the required properties) and a physical rename expression is added. Afterwards, the selection can be applied, so the physical expression is added, the new properties are calculated and both are stored in the hash table.

The reconstruction is similar for binary operators like joins: Both input plans are examined recursively, renames are added as needed and the two expressions are combined using a physical operator.

```
NESTEDLOOP::BUILDALGEBRA($plan$)
 1  $(algebra, properties) \leftarrow reconstructionTable[plan]$
 2  if $(algebra, properties)$ is undefined
 3     then $left \leftarrow plan.left.rule.$BUILDALGEBRA$(plan.left)$
 4          $leftp \leftarrow reconstructionTable[plan.left].properties$
 5          if $requiredLeft \not\subseteq leftp$
 6             then
 7                  $np \leftarrow$ rename $leftp$ by checking the operators in $plan.left$
 8                  $left \leftarrow$ add new rename operator $leftp \rightarrow np$ to $left$
 9                  $leftp \leftarrow np$
10          $right \leftarrow plan.right.rule.$BUILDALGEBRA$(plan.right)$
11          $rightp \leftarrow reconstructionTable[plan.right].properties$
```

```
12            if $requiredRight \nsubseteq rightp$
13              then
14                    $np \leftarrow$ rename $rightp$ by checking the operators in $plan.right$
15                    $right \leftarrow$ add new rename operator $rightp \rightarrow np$ to $right$
16                    $rightp \leftarrow np$
17            $algebra \leftarrow$ construct $left \bowtie^{NL} right$
18            $properties \leftarrow leftp \cup rightp \cup produced$
19            $reconstructionTable[plan] \leftarrow (algebra, properties)$
20  return $algebra$
```

7. Rules

7.1. Introduction

In Chapter 5 we have described the architecture of a plan generator. However, the plan generator is incomplete without at least some basic rules for query optimization. In this chapter we discuss the most common rules relevant for e.g. SQL queries. After providing an overview of the rules, we look at rule instantiation and extend the plan generator interface a bit to make the rules more convenient to write. Afterwards, we discuss the different operator rules, from the simple to the more complex. Finally, we consider how complex algebraic equivalences can be modeled.

The rule class hierarchy is shown in Figure 7.1. The rules are organized in three groups: Rules derived from *BaseRule* are rules for access paths that can be constructed before the search phase, while rules derived from *SearchRule* are used during the search phase to construct partial plans. The rules derived from *HelperRule* (which is only an empty marker class) are used only indirectly by the plan generator. For example, the *Sort* rule is used by multiple other rules (e.g. in *Join* for a sort merge join), but not directly by the plan generator.

Note that the rules are presented independently of each other, but they are not really independent. For example, in general a selection can be pushed down the left-hand side of a left outer join, but not the right-hand side. Here we assume that the preparation step identifies the situation where a push down is not possible and forces the selection above the outer join. This is done by including the outer join as a requirement for the selection, see Section 7.6 for details. But while such dependencies can be described quite easily, detecting such operator dependencies requires that the rules know about each other. This is unfortunate, as it severely limits the extensibility.

For the small rule set presented here, this problem is not so severe, but a more general implementation should try to solve this. One possibility would be to provide a formal specification for each rule, including information about associativity, commutativity, linearity, specific equivalences, etc. This would allow a formal reasoning about the rules and make the rule system more easily extensible. However, this approach is not always possible, as rules need not correspond to algebraic operators. In particular, rule can change their behavior depending on the operators present in their input (see Section 7.6 for an example). In this case, it makes more sense to group rules depending on their behavior (using an appropriate class hierarchy) and base the reasoning upon the most specific class "known" to a rule. While this might miss some optimization possibilities if a rule is now known by another, it offers a flexible and extensible way to handle rule dependencies.

Alternatively, the rule could be organized in a different way: Instead of mod-

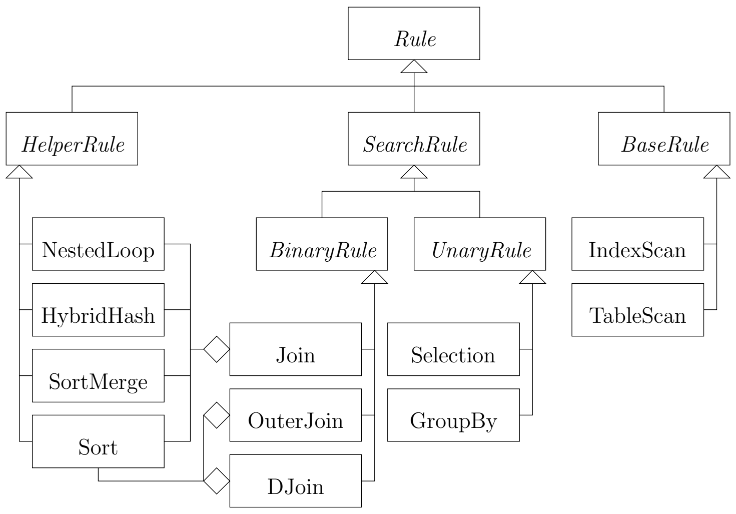

Figure 7.1.: Rule hierarchy

eling the rules after the algebraic operators, they could be modeled after parts of the query. For example, instead of using one rule per join, the plan generator could use one rule per predicate. The rule could then decide if it creates a join plan, a selection plan etc. The dependencies between the rules could be derived directly from the query representation (e.g. a join graph). However, formulating such rules which are also suitable for complex queries is beyond the scope of this work. Therefore, we here only consider rules which model operators.

7.2. Instantiating Rules

As stated in Section 5.7.1, all rules relevant for plan generation are instantiated during the preparation phase; but we have not given a formal description of this yet. In fact, the instantiation depends on the structure of the rules and on the query representation. The rules presented here model both physical and logical operators, i.e. there exists a rule for each supported physical operator (e.g. a sort merge join) and when there is no 1:1 mapping between logical and physical operators (e.g. for joins), there also exists a rule for the logical operator, which selects a rule for the physical operator during plan generation.

This makes rule instantiation simple if the query is represented as an expression in the logical algebra: Instantiate a rule for each logical operator. For example, the query $\sigma_{c=5}(A \bowtie_{a=b} B)$ would result in one *Selection* rule instance annotated with the selectivities etc. of $c = 5$, one *Join* rule instance annotated with the characteristics of $a = b$ and two *Scan* instances, one for A and one for B. Note that indirectly more rule instances are created: The *Join* rule represents a logical join, but has to select a physical join operator during plan generation. Therefore, creating a *Join* rule also creates rules for *NestedLoop*, *SortMerge*, *Sort* (on a and on b) etc. But these additional rules are embedded in the *Join* rule and not visible for the plan generator.

If the query is represented in a logical calculus (e.g. a query graph), rule instantiation requires some more work but is not very complex either: For example, given a join graph, instantiate a *Scan* rule for each node and a *Join* rule for each edge (respectively a *Selection* rule for each self edges). The exact rule may vary depending on the edge (e.g. non-equijoins usually require a nested loop join), but still the rules can be derived directly from the query. Some care is required for cyclic query graphs (as joins can become selections here), but this can also be solved by using smarter join rules that decide during plan generation if they are selections or joins. In this case, the predicate gives rise to the meaning of the rule; the rules no longer model algebraic operators, but parts of the query.

Supporting plan alternatives generated by rewrite rules is more difficult. It is not advisable that the rewrite rules just generate a new algebra expression which is then provided as an alternative rule set to the plan generator: While this works in the sense that the correct solution is produced, it is inefficient. The two expressions probably overlap, as the rewrite rule will not change the whole query, but only part of it. But the plan generator will not recognize this if the rules are different, and will try both rule variants independently, resulting in duplicate work. This can be solved in two ways: Either the preparation step identifies the overlap and creates the corresponding rules only once, or the rewrite rules do not create completely new algebra expressions, but annotate parts of the existing expression with alternatives. The first variant makes the rewrite rules simple, the second the preparation step. Either can be chosen, but the second one is probably preferable, as the rewrite rules have more information than available by just looking at the two different expressions (in particular, they can mark two expressions as equivalent which are only equivalent under certain constraints).

7.3. Plan Generator Interface

The basic function offered by the plan generator is

PLANGEN(*goal : bit set*) : *PlanSet*

that produces the set of plans satisfying the requested bit properties. However, some rules need not only certain bit properties, but also a certain ordering or grouping. Therefore, we add a convenience function that calls the basic PLANGEN function and ensures that the result contains a plan with the requested ordering or grouping:

```
PLANGEN($goal : bitset, ordering : Order, enforcer : Rule$)
1  $plans \leftarrow$ PLANGEN($goal$)
2  if $plans = \emptyset$
3     then return $plans$
4  if $\nexists p \in plans : p$ satisfies $ordering$
5     then $p \leftarrow$ cheapest plan in $plans$
6          $p_2 \leftarrow$ create a new $Plan$
7          $p_2.rule \leftarrow enforcer$
```

```
 8            $p_2.left \leftarrow p$
 9            $enforcer.$UPDATEPLAN$(p_2)$
10            $plans \leftarrow plans \cup \{p_2\}$
11    return $plans$
```

The extended function gets a goal, a requested ordering or grouping, and an enforcer rule that can produce the requested ordering or grouping. It first calls the basic function to get the set of plans. If this set is empty it gives up, as no plan could be found. If there are plans, but none of them satisfies the requested ordering, it creates a new plan by choosing the cheapest plan found and applying the enforcer rule (*updatePlan* calculates all statistics). Note that the notion of a cheapest plan might not be unique. It is here, as we assume that all enforcers materialize their result, but otherwise it might be necessary to create multiple plans instead of just one.

7.4. Updating Sharing Properties

To make the rules more readable, we pretend to factorize updating the sharing properties into a separate function. In a real implementation the update is trivial, but we give a more detailed high-level description here: The *sharing* attribute consists of a bit set where a bit is set if a plan is constructed using an equivalence class representative. As this is meant to identify shared subplans and only whole subplans can be shared, the bit must only be set if the input is also sharable, i.e., constructed using an equivalence class representative. Note that the plan generator did not select a representative for equivalence classes with just one entry, as then not sharing can occur during plan generation anyway. The corresponding pseudo code for unary operators is shown below:

```
UNARYOPERATOR::CALCSHARING($input : Plan$)
1  if $this$ is an equivalence class representative
2     then if $input.rule \in input.sharing$
3             then return $input.sharing \cup \{this\}$
4  return $input.sharing$
```

Binary operators are analogous, they check both inputs and union the two *sharing* attributes. Scans which are representatives directly set the bit, as then the subplan just consists of one (sharable) operator.

7.5. Rules

In the rest of this section we describe different rules. As in our approach each rule is a class that is instantiated during the preparation step, we first give the class definition, then the INITIALIZE/UPDATEPLAN method to set plan characteristics and for the search rule the SEARCH method to direct plan generation.

Note that the functions used to modify the members *state* and *cardinality* are explained in Chapter 9; they belong to the cost model.

7.5.1. Scan

The most basic rules are the rules for scans, either table scans or index scans. As they require no input, they are derived from *BaseRule* so that the plan generator can create access paths before the search phase. We only give the index scan rule here, as the table scan is even simpler and can be derived trivially from the index scan.

```
IndexScan : BaseRule
    produced : bit set
    index : segment_t
    relation : segment_t

    INITIALIZE(plans : PlanSet) : void
    UPDATEPLAN(plan : Plan) : void
```

The first method sets the attributes of the plan set to the characteristics of the relation:

```
INDEXSCAN::INITIALIZE(plans)
1  plans.properties ← produced
2  plans.state.cardinality ← cardinality of relation
3  plans.state.tupleSize ← avg. tuple size of relation
```

So it sets the *properties* member to the produced bit set calculated in the preparation phase and the *state* member to the characteristics of the relation. This would be the same for a table scan, as these are logical characteristics. Now the second method sets the physical characteristics for a concrete plan:

```
INDEXSCAN::UPDATEPLAN(plan)
1  plan.rule ← this
2  plan.sharing ← UPDATESHARING()
3  plan.ordering ← ordering of the index
4  plan.costs ← costs to scan the index
```

The ordering stored in the plan is the physical tuple ordering implied by the index structure. The representation is discussed in Chapter 8.

Of course it makes more sense to use an index scan in combination with a selection predicate. This can be done with a rule that behaves like an index scan, but sets the cost accordingly and sets both the scan and the selection in the *produced* member.

7.5.2. Sort

The rule for sort operators is also simple (at least from the point of view of the plan generator) . It is never scheduled directly by the plan generator, but only inserted on demand by other rules to enforce a required ordering:

```
Sort : HelperRule
    ordering : Order
    UPDATEPLAN($plan$ : $Plan$) : $void$
```

The member *ordering* describes the physical ordering produced by the sort operator (see Chapter 8 for a detailed discussion of *Order*). As *Sort* is a direct descendant of *Rule*, it only offers the UPDATEPLAN method. Nothing more is required, as other rules take care of the correct usage, see for example Section 7.5.4.

```
SORT::UPDATEPLAN($plan$)
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow$ UPDATESHARING($plan.left$)
3  $plan.ordering \leftarrow ordering$
4  $plan.costs \leftarrow plan.left.costs +$ costs of sorting
```

7.5.3. Selection

The first more intelligent rule is the rule for selection. It actually influences the search, although only in a very limited way. Note that we assume here that a selection can be put anywhere in the plan where the selection predicate can be evaluated. When this is not true (e.g. for queries with outer joins), the constraints are handled by the conflicting operators (see Section 7.6). Below is the class definition, the member *required* consists of all attributes required for the selection and the member *produced* consists of the fact that the selection has been applied.

```
$Selection$ : $UnaryRule$
    $produced$    : bit set
    $required$    : bit set
    $selectivity$ : $double$

    UPDATEPLAN($plan$ : $Plan$) : $void$
    SEARCH($plans$ : $PlanSet$) : $void$
```

The first method just adds the costs and changes the ordering according to the functional dependency created by the selection:

```
SELECTION::UPDATEPLAN($plan$)
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow$ UPDATESHARING($plan.left$)
3  $plan.ordering \leftarrow plan.left.ordering$ adjusted by induced FD
4  $plan.costs \leftarrow plan.left.costs +$ costs for the predicate
```

The second method first checks if the desired plans could actually be produced by the selection (this is redundant if the *filter* member is set correctly), triggers the plan generator to find the best plans without selection and then adds the selection. Note that the implementation is provided by the base class *UnaryRule*, as the search logic is the same for other unary operators (e.g. *GroupBy*).

UNARYRULE::SEARCH(*plans*)

```
if $(produced \cup required) \not\subseteq plans.properties$
  then return
$input \leftarrow$ PLANGEN$(plans.properties \setminus produced)$
if $|plans| = 0 \land |input| > 0$
  then $plans.state.cardinality \leftarrow input.state.cardinality * selectivity$
       $plans.state.tupleSize \leftarrow input.state.tupleSize$
for each $p$ in $input$
  do $p_2 \leftarrow$ create a new $Plan$
     $p_2.left \leftarrow p$
     UPDATEPLAN$(p_2)$
     $plans \leftarrow plans \cup \{p_2\}$
```

Note that in case that *plans* is empty (i.e. the first plan will be added) the selection updates the cardinality and the tuple size (lines 5-6 are in reality a small virtual method, as they are different for other operators).

7.5.4. Join

The join rule is much more complex, as it consists of multiple rules for the different join algorithms. For simplicity, we restrict ourselves to equi-joins here (theta-joins can be handled by just extending the nested-loop rule) and, as with selections, assume that the joins are freely reorderable. See the outer join rule for an example where this is not true. The join rule is a composition of multiple relatively simple rules, we look at them afterwards. Below is the class definition for equi-joins:

```
Join : BinaryRule
    produced      : bit set
    requiredLeft  : bit set
    requiredRight : bit set
    selectivity   : double
    nl            : NestedLoop
    sm            : SortMerge
    hh            : HybridHash
    orderLeft     : Order
    sortLeft      : Sort
    orderRight    : Order
    sortRight     : Sort

    UPDATEPLAN(plan : Plan) : void
    SEARCH(plans : PlanSet) : void
```

The attributes *produced*, *requiredLeft* and *requiredRight* consist of the bit properties produced and required by the join. Their union forms the *filter* of the join rule. As with selections, the selectivity is stored in an attribute with the same name. More interesting are the attributes *nl*, *sm* and *hh*: They are nested rules that are used by the main join logic to describe the different supported join operators. As the sort merge join requires that the input satisfies a certain ordering, the orderings and the corresponding enforcers are stored in *orderLeft*, *sortLeft*, *orderRight* and *sortRight*.

The UPDATEPLAN method is not really relevant for the generic join rule, as it will select a more specific rule during the search phase. So no plan should be constructed with the generic rule. If this happens for some reason, the generic join can always fall back to the nested loop:

```
JOIN::UPDATEPLAN($plan$)
1  $nl$.UPDATEPLAN($plan$)
```

The SEARCH method is more involved. After the check if the rule is really applicable (redundant if the filter is set correctly), it first determines which bit properties are not determined by the join requirements and then asks the plan generator to solve the different possible combinations. This call uses the convenience function defined above to make sure that a plan satisfying the requirements of the sort merge join is included. After the initialization of the plan properties (if this is the first plan), it uses the different join rules to construct plan alternatives.

```
JOIN::SEARCH($plans$)
 1  if ($produced \cup requiredLeft \cup requiredRight$) $\not\subseteq plans.properties$
 2    then return
 3  $space \leftarrow plans.properties \setminus (produced \cup requiredLeft \cup requiredRight)$
 4  for each $lp \subseteq space$
 5    do $rp \leftarrow space \setminus lp$
 6       $lplans \leftarrow$ PLANGEN($requiredLeft \cup lp, orderLeft, sortLeft$)
 7       if $|lplans| = 0$
 8         then continue
 9       $rplans \leftarrow$ PLANGEN($requiredRight \cup rp, orderRight, sortRight$)
10       if $|rplans| = 0$
11         then continue
12       if $|plans| = 0$
13         then $plan.state =$ JOIN($lplan.state, rplan.state, cardinality$)
14       for each $l$ in $lplans$
15         do for each $r$ in $rplans$
16              do if $l$ satisifies $orderLeft \wedge r$ satisfies $orderRight$
17                   then $p \leftarrow$ create a new $Plan(l, r)$
18                        $sm$.UPDATEPLAN($p$)
19                        $plans \leftarrow plans \cup \{p\}$
20                 $p \leftarrow$ create a new $Plan(l, r)$
21                 $nl$.UPDATEPLAN($p$)
22                 $plans \leftarrow plans \cup \{p\}$
23                 $p \leftarrow$ create a new $Plan(l, r)$
24                 $hh$.UPDATEPLAN($p$)
25                 $plans \leftarrow plans \cup \{p\}$
```

The different join rules are all direct descendants of *Rule* (via the empty marker class *Helper*), as they do not have to manage the search phase themselves. The general structure is:

```
$JoinAlgorithm : HelperRule$
    $join : Join$

    $\text{UPDATEPLAN}(plan : Plan) : void$
```

So the join algorithms do not store any information themselves, but use the data of the enclosing generic join rule.

Nested Loop

The nested loop is the simplest join algorithm, it iterates over the right-hand side for each tuple on the left-hand side. It can be used for any kind of join predicate.

```
$\text{NESTEDLOOP::UPDATEPLAN}(plan)$
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow \text{UPDATESHARING}(plan.left, plan.right)$
3  $plan.ordering \leftarrow plan.left.ordering$ adjusted by induced FD
4  $plan.costs \leftarrow \text{INPUTCOSTS}(plan.left, 1, plan.right, plan.left.card)$
5  $plan.costs \leftarrow plan.costs +$ costs for the predicate
```

Sort Merge

The sort merge join can only be used for equi-joins and requires that the input is ordered on the join attributes. Note that this is guaranteed by the generic join rule: The sort merge join *updatePlan* method is only called in this case.

```
$\text{SORTMERGE::UPDATEPLAN}(plan)$
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow \text{UPDATESHARING}(plan.left, plan.right)$
3  $plan.ordering \leftarrow plan.left.ordering$ adjusted by induced FD
4  $plan.costs \leftarrow \text{INPUTCOSTS}(plan.left, 1, plan.right, 1)$
5  $plan.costs \leftarrow plan.costs +$ costs for the predicate
```

Hybrid Hash

The hybrid hash join is also only usable for equi-joins and has a cost function that depends on many parameters, especially the available memory. For a detailed discussion see [65], here we just assume that the costs can be calculated somehow. Note that the hybrid hash join destroys the ordering of the input.

```
$\text{HYBRIDHASH::UPDATEPLAN}(plan)$
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow \text{UPDATESHARING}(plan.left, plan.right)$
3  $plan.ordering \leftarrow$ no ordering
4  $plan.costs \leftarrow$ costs for the hybrid hash join
```

7.5.5. Outer Join

In contrast to the regular join, the outer join is not freely reorderable. This means that in many cases changing the order of operators (e.g. an outer join and a selection) influences the semantic of the query. The simplest solution for this is to "fix" the position of the outer join in relation to the other operators: Add all operators that must come before to the requirements of the outer join and let the operators that must come afterwards require the outer join. The operator below and above the outer join can still be optimized, but they cannot be moved across the outer join, guaranteeing the correct semantics.

However, this fixing of the position misses some optimization opportunities. For example, a selection can be pushed down the left side of a left outer join, and also the join ordering can be changed sometimes. See [16] for a detailed discussion of outer join optimization. These optimizations can be done in two ways: Either plan alternatives can be considered (see Section 5.8 for an example), or for simpler optimizations like the exchange of selections and outer joins it is sufficient to relax the operator dependencies accordingly (see Section 7.6). Of course this should only be allowed if the selection is only applicable in the left-hand side of the outer join, but this can be checked during the preparation phase.

These semantic constraints aside, the outer join rule is nearly identical to the other join rules. Note that the outer join will also offer a *search* method like the regular rule, but as it is nearly identical, we omit it here. The rules are so similar that it might be useful to specify only one rule and use a flag to discern between regular and outer join, but we assume a special rule for a hash-based outer join operator here.

```
OuterJoin::updatePlan($plan$)
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow$ updateSharing($plan.left, plan.right$)
3  $plan.ordering \leftarrow$ no ordering
4  $plan.costs \leftarrow$ costs for the outer join
```

7.5.6. DJoin

Dependent joins are joins where the evaluation of one side depends on the current value of the other. Thus, dependent joins are not commutative, but associative and otherwise reorderable similar to normal joins. The free variables of the dependent side induce additional constraints, they are modelled as requirements of the operator itself (see Section 7.6 for a detailed example). The runtime behavior of the djoin usually degenerates into a nested loop join, but for equi-joins some optimizations are possible if the left-hand side is grouped on the join attribute. As the difference to the regular join is somewhat larger, we give the full rule definition here:

```
DJoin : BinaryRule
    produced       : bit set
    requiredLeft   : bit set
```

requiredRight	: bit set
selectivity	: *double*
equijoin	: *boolean*
groupingLeft	: *Order*
groupLeft	: *Sort*

```
UPDATEPLAN($plan$ : $Plan$) : $void$
SEARCH($plans$ : $PlanSet$) : $void$
```

Note that although the enforcer *groupLeft* was declared as *Sort*, a hash-based grouping operator would be sufficient, if available. The update method is just like a method for a regular nested loop, but takes into account the grouping property, if available:

```
DJoin::UPDATEPLAN($plan$)
$plan.rule \leftarrow this$
$plan.sharing \leftarrow$ UPDATESHARING($plan.left, plan.right$)
$plan.ordering \leftarrow plan.left.ordering$ adjusted by induced FD
if $plan.left.ordering$ satisfy $groupingLeft \wedge equijoin =$ true
  then $plan.costs \leftarrow$ INPUTCOSTS($plan.left, 1, plan.right,$ number of groupings )
  else $plan.costs \leftarrow$ INPUTCOSTS($plan.left, 1, plan.right, plan.left.cardinality$)
$plan.costs \leftarrow plan.costs +$ costs for the predicate
```

The *search* method is similar to the regular join, but tries to group the left-hand side first if the djoin is an equijoin

```
DJoin::SEARCH($plans$)
if ($produced \cup requiredLeft \cup requiredRight$) $\not\subseteq plans.properties$
  then return
$space \leftarrow plans.properties \setminus (produced \cup requiredLeft \cup requiredRight)$
for each $lp \subseteq space$
  do $rp \leftarrow space \setminus lp$
     if $equijoin =$ true
       then $lplans \leftarrow$ PLANGEN($requiredLeft \cup lp, groupingLeft, groupLeft$)
       else $lplans \leftarrow$ PLANGEN($requiredLeft \cup lp$)
     if $|lplans| = 0$
       then continue
     $rplans \leftarrow$ PLANGEN($requiredRight \cup rp, orderRight, sortRight$)
     if $|rplans| = 0$
       then continue
     if $|plans| = 0$
       then $plan.state =$ JOIN($lplan.state, rplan.state, cardinality$)
     for each $l$ in $lplans$
       do for each $r$ in $rplans$
            do $p \leftarrow$ create a new $Plan(l, r)$
               UPDATEPLAN($p$)
               $plans \leftarrow plans \cup \{p\}$
```

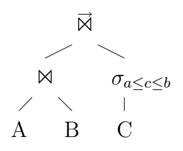

Figure 7.2.: Simple dependent join example

7.5.7. Group By

Like the outer join the group-by operator is not reorderable in general. However, several optimization techniques have been proposed, most of them require using plan alternatives. See [80, 81] for a detailed discussion and Section 7.6 for an example. The rule shown below considers two different implementations, either a hash-based group-by or re-using an already grouped input.

```
$GroupBy$ : $UnaryRule$
    $produced$    : bit set
    $required$    : bit set
    $selectivity$ : $double$
    $grouping$    : $Order$

    UPDATEPLAN($plan$ : $Plan$) : $void$
    SEARCH($plans$ : $PlanSet$) : $void$
```

```
GROUPBY::UPDATEPLAN($plan$)
1  $plan.rule \leftarrow this$
2  $plan.sharing \leftarrow$ UPDATESHARING($plan.left$)
3  if $plan.left.ordering$ satisfies $grouping$
4     then $plan.ordering \leftarrow plan.left.ordering$
5          $plan.costs \leftarrow plan.left.costs +$ costs for the predicate
6     else $plan.ordering \leftarrow grouping$
7          $plan.costs \leftarrow plan.left.costs +$ costs for grouping
```

The *search* method is equivalent to the search method of *Selection* (in fact, it is provided by the common base class *UnaryRule*).

7.6. Operator Dependencies

As stated above, dependencies are used to model algebraic equivalences. We will now consider some more complex equivalences to illustrate the approach for the given rule set. Note that we assume that all rules produce at least the bit properties "operator applied" (for the corresponding logical operator) and "attribute available" (if they produce new attributes).

Rules for operators that are freely reorderable (especially *Selection* and *Join*) only require bit properties according to their syntax constraints. For dependent joins addition constraints are already required. Consider the operator tree shown in Figure 7.2. In fact, the tree could be changed trivially to use only

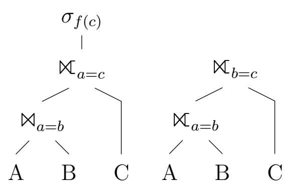

Figure 7.3.: Two examples of outer joins

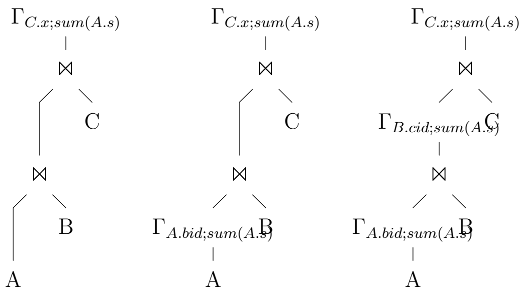

Figure 7.4.: Coalescing grouping example

regular joins, but it is useful as an illustration. The scans A and B require nothing and produce the properties a and b (we assume that the scans produce only one attribute). The regular join between A and B requires the attributes a and b. Note that while the attributes a and b are "available" in the dependent branch, they must be treated differently (otherwise, predicates could migrate from the independent to the dependent branch, which is not correct in general). Therefore, we rename a and b (which are free attributes of the dependent branch) into a' and b'. Thus, the scan C produces the attribute c and also a' and b' (as they are always available). The selection now requires a', b' and c. Finally, the dependent join requires a and b on the left-hand side and c on the right-hand side.

A simple example for outer joins is shown on the left-hand side of Figure 7.3. Here the scans A, B and C produce the attributes a, b and c; the join requires a on its left-hand side and b on its right-hand side. The outer join requires a on its left-hand side and c on its right-hand side, so the joins are reorderable (within the syntax constraints). The selection only requires c syntactically, but it must not be evaluated before the outer join. Therefore, the outer join is also included in the requirements of the selection, guaranteeing the correct execution sequence. A more complex join situation is shown on the right-hand side of Figure 7.3. The first outer join requires a and b, while the second outer join requires b and c. But the first outer join must be executed before the second one (in particular, it is not allowed to push the second join down the right-hand side of the first join). Therefore, the second join must "require" the first join on its left-hand side.

Similar to outer joins, group-bys are not freely reorderable in general, the same techniques as discussed above can be used to guarantee a certain operator ordering. But in addition to this, group-by operators allow an interesting optimization technique called coalescing grouping [8]. An example for this is shown in Figure 7.4. Given suitable join and group-by predicates, the group-by can be duplicated (or, in some cases, just moved) below a join, thus reducing the costs of the join itself. In the worst case with n joins, there are 2^n such grouping possibilities, which makes it inefficient to model the problem using plan alternatives, as increasing the number of rules by a factor of 2^n greatly increases the search time. This could be done by the group-by rule instead: The rule asks the plan generator to produce plans with a given set of bit properties. Therefore, it could explicitly request plans with an additional group-by at a certain position (the group-by could be added by an extended rule for joins, for example). This would still increase the search space by 2^n (ignoring pruning), but the rule set remains compact and, thus, checking for applicable rules remains cheap.

8. Orderings and Groupings

8.1. Motivation

The most expensive operations (e.g. join, grouping, duplicate elimination) during query evaluation can be performed more efficiently if the input is ordered or grouped in a certain way. Therefore, it is crucial for query optimization to recognize cases where the input of an operator satisfies the ordering or grouping requirements needed for a more efficient evaluation. Since a plan generator typically considers millions of different plans – and, hence, operators –, this recognition easily becomes performance critical for query optimization, often leading to heuristic solutions.

The importance of exploiting available orderings has been recognized in the seminal work of Selinger et al [69]. They presented the concept of interesting orderings and showed how redundant sort operations could be avoided by reusing available orderings, rendering sort-based operators like sort-merge join much more interesting.

Along these lines, it is beneficial to reuse available grouping properties, for example for hash-based operators. While heuristic techniques to avoid redundant group-by operators have been given [8], groupings have not been treated as thoroughly as orderings. One reason might be that while orderings and groupings are related (every ordering is also a grouping), groupings behave somewhat differently. For example, a tuple stream grouped on the attributes $\{a, b\}$ need not be grouped on the attribute $\{a\}$. This is different from orderings, where a tuple stream ordered on the attributes (a, b) is also ordered on the attribute (a). Since no simple prefix (or subset) test exists for groupings, optimizing groupings even in a heuristic way is much more difficult than optimizing orderings. Still, it is desirable to combine order optimization and the optimization of groupings, as the problems are related and treated similarly during plan generation. Recently, some work in this direction has been published [77]. However, this only covers a special case of grouping, as we will discuss in some detail in Section 8.3.

Existing frameworks usually consider only order optimization, and experimental results have shown that the costs for order optimization can have a large impact on the total costs of query optimization[62]. Therefore, some care is needed when adding groupings to order optimization, as a slowdown of plan generation would be unacceptable.

In this chapter, we present a framework to efficiently reason about orderings and groupings. It can be used for the plan generator described in Chapter 5, but is actually an independent component that could be used in any kind of plan generator. Experimental results show that it efficiently handles orderings and groupings at the same time, with no additional costs during plan generation and only modest one time costs. Actually, the operations needed for both ordering and grouping optimization during plan generation can be performed in $O(1)$, basically allowing to exploit groupings for free. Parts of this chapter were previously published in [60, 61, 62].

8.2. Problem Definition

The order manager component used by the plan generator combines order optimization and the handling of grouping in one consistent set of algorithms and data structures. In this section, we give a more formal definition of the problem and the scope of the framework. First, we define the operations of ordering and grouping (Section 8.2.1 and 8.2.2). Then, we briefly discuss functional dependencies (Section 8.2.3) and how they interact with algebraic operators (Section 8.2.4). Finally, we explain how the component is actually used during plan generation (Section 8.2.5).

8.2.1. Ordering

During plan generation, many operators require or produce certain orderings. To avoid redundant sorting, it is required to keep track of the orderings a certain plan satisfies. The orderings that are relevant for query optimization are called *interesting orders* [69]. The set of *interesting orders* for a given query consists of

1. all orderings required by an operator of the physical algebra that may be used in a query execution plan for the given query, and

2. all orderings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

This includes the final ordering requested by the given query, if this is specified.

The interesting orders are *logical orderings.* This means that they specify a condition a tuple stream must meet to satisfy the given ordering. In contrast, the *physical ordering* of a tuple stream is the actual succession of tuples in the stream. Note that while a tuple stream has only one physical ordering, it can satisfy multiple logical orderings. For example, the stream of tuples $((1,1),(2,2))$ with schema (a,b) has one physical ordering (the actual stream), but satisfies the logical orderings a, b, ab and ba.

Some operators, like `sort`, actually influence the physical ordering of a tuple stream. Others, like `select`, only influence the logical ordering. For example, a `sort[a]` produces a tuple stream satisfying the ordering (a) by actually changing the physical order of tuples. After applying `select[a=b]` to this tuple stream, the result satisfies the logical orderings $(a),(b),(a,b),(b,a)$, although the physical ordering did not change. Deduction of logical orderings can be described by using the well-known notion of *functional dependency* (FD) [70]. In general, the influence of a given algebraic operator on a set of logical orderings can be described by a set of functional dependencies.

We now formalize the problem. Let $R = (t_1, \ldots, t_r)$ be a stream (ordered sequence) of tuples in attributes $A_1, \ldots, A_n$. Then R *satisfies the logical ordering* $o = (A_{o_1}, \ldots, A_{o_m})$ $(1 \leq o_i \leq n)$ if and only if for all $1 \leq i < j \leq r$ the following condition holds:

$$\begin{array}{llll} & (t_i.A_{o_1} \leq t_j.A_{o_1}) & & \\ \wedge & \forall 1 < k \leq m & & (\exists 1 \leq l < k(t_i.A_{o_l} < t_j.A_{o_l})) \vee \\ & & & ((t_i.A_{o_{k-1}} = t_j.A_{o_{k-1}}) \wedge \\ & & & (t_i.A_{o_k} \leq t_j.A_{o_k})) \end{array}$$

Next, we need to define the inference mechanism. Given a logical ordering $o = (A_{o_1}, \ldots, A_{o_m})$ of a tuple stream R, then R obviously satisfies any logical ordering that is a prefix of o including o itself.

Let R be a tuple stream satisfying both the logical ordering $o = (A_1, \ldots, A_n)$ and the functional dependency $f = B_1, \ldots, B_k \rightarrow B_{k+1}$[1] with $B_i \in \{A_1 \ldots A_n\}$. Then R also satisfies any logical ordering derived from o as follows: add B_{k+1} to o at any position such that all of $B_1, \ldots, B_k$ occurred before this position in o. For example, consider a tuple stream satisfying the ordering (a, b); after inducing the functional dependency $a, b \rightarrow c$, the tuple stream also satisfies the ordering (a, b, c), but not the ordering (a, c, b). Let O' be the set of all logical orderings that can be constructed this way from o and f after prefix closure. Then, we use the following notation: $o \vdash_f O'$. Let e be the equation $A_i = A_j$. Then, $o \vdash_e O'$, where O' is the prefix closure of the union of the following three sets. The first set is O_1 defined as $o \vdash_{A_i \rightarrow A_j} O_1$, the second is O_2 defined as $o \vdash_{A_j \rightarrow A_i} O_2$, and the third is the set of logical orderings derived from o where a possible occurrence of A_i is replaced by A_j or vice versa. For example, consider a tuple stream satisfying the ordering (a); after inducing the equation $a = b$, the tuple stream also satisfies the orderings $(a, b), (b)$ and (b, a). Let e be an equation of the form $A = \mathit{const}$. Then O' $(o \vdash_e O')$ is derived from o by inserting A at any position in o. This is equivalent to $o \vdash_{\emptyset \rightarrow A} O'$. For example, consider a tuple stream satisfying the ordering (a, b); after inducing the equation $c = \mathit{const}$ the tuple stream also satisfies the orderings $(c, a, b), (a, c, b)$ and (a, b, c).

Let O be a set of logical orderings and F a set of functional dependencies (and possibly equations). We define the sets of inferred logical orderings $\Omega_i(O, F)$ as follows:

$$\begin{array}{rcl} \Omega_0(O, F) & := & O \\ \Omega_i(O, F) & := & \Omega_{i-1}(O, F) \cup \\ & & \bigcup\limits_{f \in F, o \in \Omega_{i-1}(O,F)} O' \text{ with } o \vdash_f O' \end{array}$$

Let $\Omega(O, F)$ be the prefix closure of $\bigcup_{i=0}^{\infty} \Omega_i(O, F)$. We write $o \vdash_F o'$ if and only if $o' \in \Omega(O, F)$.

[1] Any functional dependency which is not in this form can be normalized into a set of FDs of this form.

8.2.2. Grouping

It was shown in [77] that, similar to order optimization, it is beneficial to keep track of the groupings satisfied by a certain plan. Traditionally, group-by operators are either applied after the rest of the query has been processed or are scheduled using some heuristics [8]. However, the plan generator could take advantage of grouping properties produced e.g. by avoiding re-hashing if such information was easily available.

Analogous to order optimization, we call this *grouping optimization* and define that the set of *interesting groupings* for a given query consists of

1. all groupings required by an operator of the physical algebra that may be used in a query execution plan for the given query

2. all groupings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

This includes the grouping specified by the group-by clause of the query, if any exists.

These groupings are similar to logical orderings, as they specify a condition a tuple stream must meet to satisfy a given grouping. Likewise, functional dependencies can be used to infer new groupings.

More formally, a tuple stream $R = (t_1, \dots, t_r)$ in attributes $A_1, \dots, A_n$ *satisfies the grouping* $g = \{A_{g_1} \dots, A_{g_m}\}$ $(1 \leq g_i \leq n)$ if and only if for all $1 \leq i < j < k \leq r$ the following condition holds:

$$\begin{array}{rl} & \forall 1 \leq l \leq m \quad t_i.A_{g_l} = t_k.A_{g_l} \\ \Rightarrow & \forall 1 \leq l \leq m \quad t_i.A_{g_l} = t_j.A_{g_l} \end{array}$$

Two remarks are in order here. First, note that a grouping is a set of attributes and not – as orderings – a sequence of attributes. Second, note that given two groupings g and $g' \subset g$ and a tuple stream R satisfying the grouping g, R need not satisfy the grouping g'. For example, the tuple stream $((1,2),(2,3),(1,4))$ with the schema (a,b) is grouped by $\{a,b\}$, but not by $\{a\}$. This is different from orderings, where a tuple stream satisfying an ordering o also satisfies all orderings that are a prefix of o.

New groupings can be inferred by functional dependencies as follows: Let R be a tuple stream satisfying both the grouping $g = \{A_1, \dots, A_n\}$ and the functional dependency $f = B_1, \dots, B_k \rightarrow B_{k+1}$ with $\{B_1, \dots, B_k\} \subseteq \{A_1, \dots, A_n\}$. Then R also satisfies the grouping $g' = \{A_1, \dots, A_n\} \cup \{B_{k+1}\}$. Let G' be the set of all groupings that can be constructed this way from g and f. Then we use the following notation: $g \vdash_f G'$. For example $\{a,b\} \vdash_{a,b \rightarrow c} \{a,b,c\}$. Let e be the equation $A_i = A_j$. Then $g \vdash_e G'$ where G' is the union of the following three sets. The first set is G_1 defined as $g \vdash_{A_i \rightarrow A_j} G_1$, the second is G_2 defined as $g \vdash_{A_j \rightarrow A_i} G_2$, and the third is the set of groupings derived from g where a possible occurrence of A_i is replaced by A_j or vice versa. For example, $\{a,b\} \vdash_{b=c} \{a,c\}$. Let e be an equation of the form $A = \mathit{const}$. Then $g \vdash_e G'$ is defined as $g \vdash_{\emptyset \rightarrow A} G'$. For example, $\{a,b\} \vdash_{c=const} \{a,b,c\}$.

Let G be a set of groupings and F be a set of functional dependencies (and possibly equations). We define the set of inferred groupings $\Omega_i(G,F)$ as follows:

$$\begin{array}{rcl} \Omega_0(G,F) & := & G \\ \Omega_i(G,F) & := & \Omega_{i-1}(G,F) \cup \\ & & \displaystyle\bigcup_{f\in F, g\in\Omega_{i-1}(G,F)} G' \text{ with } g \vdash_f G' \end{array}$$

Let $\Omega(G,F)$ be $\bigcup_{i=0}^{\infty} \Omega_i(G,F)$. We write $g \vdash_F g'$ if and only if $g' \in \Omega(G,F)$.

8.2.3. Functional Dependencies

The reasoning about orderings and groupings assumes that the set of functional dependencies is known. The process of gathering the relevant functional dependencies is described in detail in [70]. Predominantly, there are three sources of functional dependencies:

1. key constraints
2. join predicates
3. filter predicates
4. simple expressions

However, the algorithm makes no assumption about the functional dependencies. If for some reason an operator induces another kind of functional dependency (e.g., when using TID-based optimizations [53]), this can be handled the same way.

8.2.4. Algebraic Operators

To illustrate the propagation of orderings and groupings during query optimization, we give some rules for concrete (physical) operators in Figure 8.1. As a shorthand, we use the following notation:

$O(R)$	set of logical orderings and groupings satisfied by the physical ordering of the relation R
$O(S)$	inferred set of logical orderings and groupings satisfied by the tuple stream S
$x\downarrow$	$\{y \mid y \in x\}$

Note that these rules somewhat depend on the actual implementation of the operators, e.g. a blockwise nested loop join might actually destroy the ordering if the blocks are stored in hash tables. The rules are also simplified: For example, a group-by will probably compute some aggregate functions, inducing new functional dependencies. Furthermore, additional information can be derived from schema information: If the right-hand side of a dependent join (index nested loop joins are similar) produces at most one tuple, and the left-hand side is grouped on the free attributes of the right-hand side (e.g. if they do not

operator	requires	produces
scan(R)	-	$O(R)$
indexscan(Idx)	-	$O(Idx)$
map(S,$a = f(b)$)	-	$\Omega(O(S), b \rightarrow a)$
select(S,$a = b$)	-	$\Omega(O(S), a = b)$
bnl-join(S_1,S_2)	-	$O(S_1)$
indexnl-join(S_1,S_2)	-	$O(S_1)$
djoin(S_1,S_2)	-	$O(S_1)$
sort(S,$a_1, \ldots, a_n$)	-	$(a_1, \ldots, a_n)$
group-by(S,$a_1, \ldots, a_n$)	-	$\{a_1, \ldots, a_n\}$
hash(S,$a_1, \ldots, a_n$)	-	$\{a_1, \ldots, a_n\}$
sort-merge(S_1,S_2,$\vec{a} = \vec{b}$)	$\vec{a} \in O(S_1) \wedge \vec{b} \in O(S_2)$	$\Omega(O(S_1), \vec{a} = \vec{b})$
hash-join(S_1,S_2,$\vec{a} = \vec{b}$)	$\vec{a} \downarrow\in O(S_1) \wedge \vec{b} \downarrow\in O(S_2)$	$\Omega(O(S_1), \vec{a} = \vec{b})$

Figure 8.1.: Propagation of orderings and groupings

contain duplicates) the output is also grouped on the attributes of the right-hand side. This situation is common, especially for index nested loop joins, and is detected automatically if the corresponding functional dependencies are considered. Therefore, it is important that all operators consider all functional dependencies they induce.

8.2.5. Plan Generation

To exploit available logical orderings and groupings, the plan generator needs access to the combined order optimization and grouping component, which we describe as an *abstract data type* (ADT). An instance of this abstract data type `OrderingGrouping` represents a set of logical orderings and groupings, and wherever necessary, an instance is embedded into a plan note. The main operations the abstract data type `OrderingGrouping` must provide are

1. a constructor for a given logical ordering or grouping,
2. a membership test (called `containsOrdering(LogicalOrdering)`) which tests whether the set contains the logical ordering given as parameter,
3. a membership test (called `containsGrouping(Grouping)`) which tests whether the set contains the grouping given as parameter, and
4. an inference operation (called `infer(set<FD>)`). Given a set of functional dependencies and equations, it computes a new set of logical orderings and groupings a tuple stream satisfies.

These operations can be implemented by using the formalism described before: `containsOrdering` tests for $o \in O$, `containsGrouping` tests for $o \in G$ and `infer(F)` calculates $\Omega(O, F)$ respectively $\Omega(G, F)$. Note that the intuitive approach to explicitly maintain the set of all logical orderings and groupings is not useful in practice. For example, if a sort operator sorts a tuple stream on (a, b), the result is compatible with logical orderings $\{(a, b), (a)\}$. After a selection operator with selection predicate $x = const$ is applied, the set of logical orderings changes to $\{(x, a, b),\ (a, x, b),\ (a, b, x),\ (x, a),\ (a, x),\ (x)\}$. Since the size of the set increases quadratically with every additional selection predicate of the form $v = const$, a naive representation as a set of logical orderings is problematic. This led Simmen et al. to introduce a more concise representation, which is discussed in the related work section. Note that Simmen's technique is not easily applicable to groupings, and no algorithm was proposed to efficiently maintain the set of available groupings. The order optimization component described here closes this gap by supporting both orderings and groupings. The problem of quadatic growth is avoided by only implicitly representing the set. Before presenting our approach, let us discuss the existing literature in detail.

8.3. Related Work

Very few papers exist on order optimization. While the problem of optimizing interesting orders was already introduced by Selinger et al. [69], later papers usually concentrated on exploiting, pushing down or combining orders, not on the abstract handling of orders during query optimization.

A more recent paper by Simmen et al. [70] introduced a framework based on functional dependencies for reasoning about orderings. Since this is the only paper which really concentrates on the abstract handling orders and our approach is similar in the usage of functional dependencies, we will describe their approach in some more detail.

For a plan node they keep just a single (physical) ordering. Additionally, they associate all the applicable functional dependencies with a plan node. Hence, the lower-bound space requirement for this representation is essentially $\Omega(n)$, where n is the number of functional dependencies derived from the query. Note that the set of functional dependencies is still (typically) much smaller than the set of all logical orderings. In order to compute the function `containsOrdering`, Simmen et al. apply a *reduction algorithm* on both the ordering associated with a plan node and the ordering given as an argument to `containsOrdering`. Their reduction roughly does the opposite of deducing more orderings using functional dependencies. Let us briefly illustrate the reduction by an example. Assume the physical ordering a tuple stream satisfies is (a), and the required ordering is (a, b, c). Further assume that there are two functional dependencies available: $a \rightarrow b$ and $a, b \rightarrow c$. The reduction algorithm is performed on both orderings. Since (a) is already minimal, nothing changes. Let us now reduce (a, b, c). We apply the second functional dependency first. Using $a, b \rightarrow c$, the reduction algorithm yields (a, b) because c appears in (a, b, c) after a and b. Hence, c is removed. In general, every occurrence of an attribute on the right-hand side of a functional dependency is removed if all attributes of the left-hand side of the functional dependency precede the occurrence. Reduction of (a, b) by $a \rightarrow b$ yields (a). After both orderings are reduced, the algorithm tests whether the reduced required ordering is a prefix of the reduced physical ordering. Note that if we applied $a \rightarrow b$ first, then (a, b, c) would reduce to (a, c) and no further reduction would be possible. Hence, the rewrite system induced by their reduction process is not confluent. This problem is not mentioned by Simmen et al., but can have the effect that `containsOrdering` returns *false* whereas it should return *true*. The result is that some orderings remain unexploited; this could be avoided by maintaining a minimal set of functional dependencies, but the computation costs would probably be prohibitive. This problem does not occur with our approach. On the complexity side, every functional dependency has to be considered by the reduction algorithm at least once. Hence, the lower time bound is $\Omega(n)$.

In case all functional dependencies are introduced by a single plan node and all of them have to be inserted into the set of functional dependencies associated with that plan node, the lower bound for `inferNewLogicalOrderings` is also $\Omega(n)$.

Overall, Simmen et al. proposed the important framework for order optimization utilizing functional dependencies and nice algorithms to handle orderings during plan generation, but the space and time requirements are unfortunate since plan generation might generate millions of subplans. Also note that the reduction algorithm is not applicable for groupings (which, of course, was never intended by Simmen): Given the grouping $\{a, b, c\}$ and the functional dependencies $a \to b$ and $b \to c$, the grouping would be reduced to $\{a, c\}$ or to $\{a\}$, depending on the order in which the reductions are performed. This problem does not occur with orderings, as the attributes are sorted and can be reduced back to front.

A recent paper by Wang and Cherniack [77] presented the idea of combining order optimization with the optimization of groupings. Based upon Simmen's framework, they annotated each attribute in an ordering with the information whether it is actually ordered by or grouped by. For a single attribute a, they write $O_{a^O}(R)$ to denote that R is ordered by a, $O_{a^G}(R)$ to denote that R is grouped by a and $O_{a^O \to b^G}$ to denote that R is first ordered by a and then grouped by b (within blocks of the same a value). Before checking if a required ordering or grouping is satisfied by a given plan, they use some inference rules to get all orderings and groupings satisfied by the plan. Basically, this is Simmen's reduction algorithm with two extra transformations for groupings. In their paper the check itself is just written as $\in$, however, at least one reduction on the required ordering would be needed for this to work (and even that would not be trivial, as the stated transformations on groupings are ambiguous). The promised details in the cited technical report are currently not available, as the report has not appeared yet. Also note that, as explained above, the reduction approach is fundamentally not suited for groupings. In Wang's and Cherniack's paper, this problem does not occur, as they only look at a very specialized kind of grouping: As stated in their Axiom 3.6, they assume that a grouping $O_{a^G \to b^G}$ is first grouped by a and then (within the block of tuples with the same a value) grouped by b. However, this is a very strong condition that is usually not satisfied by a hash-based grouping operator. Therefore, their work is not general enough to capture the full functionality offered by a state-of-the-art query execution engine.

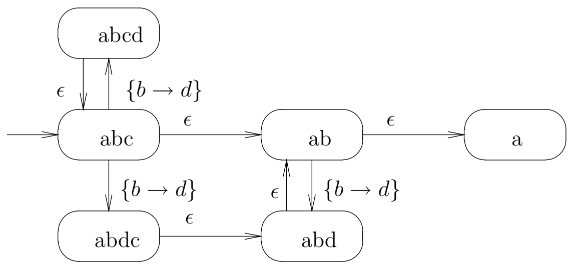

Figure 8.2.: Possible FSM for orderings

8.4. Idea

As we have seen, explicit maintenance of the set of logical orderings and groupings can be very expensive. However, the ADT `OrderingGrouping` required for plan generation does not need to offer access to this set: It only allows to test if a given interesting order or grouping is in the set and changes the set according to new functional dependencies. Hence, it is *not* required to explicitly represent this set; an implicit representation is sufficient as long as the ADT operations can be implemented atop of it. In other words, we need not be able to reconstruct the set of logical orderings and groupings from the state of the ADT. This gives us room for optimizations.

Our initial idea published in [62] was to represent sets of logical orderings as *states* of a *finite state machine* (FSM). Roughly, a state of the FSM represents a current physical ordering and the set of logical orderings that can be inferred from it given a set of functional dependencies. The edges (transitions) in the FSM are labeled by sets of functional dependencies. They lead from one state to another, if the target state of the edge represents the set of logical orderings that can be derived from the orderings the edge's source node represents by applying the set of functional dependencies the edge is labeled with. We have to use sets of functional dependencies, since a single algebraic operator may introduce more than one functional dependency.

Let us illustrate the idea by a simple example and then discuss some problems. In Figure 8.2 an FSM for the interesting order (a, b, c) and its prefixes (remember that we need prefix closure) and the set of functional dependencies $\{b \rightarrow d\}$ is given. When a physical ordering satisfies (a, b, c), it also satisfies its prefixes (a, b) and (a). This is indicated by the ϵ transitions. The functional dependency $b \rightarrow d$ allows to derive the *logical* orderings (a, b, c, d) and (a, b, d, c). This is handled by assuming that the *physical* ordering changes to either (a, b, c, d) or (a, b, d, c). Hence, these states have to be added to the FSM. We further add the transitions induced by $\{b \rightarrow d\}$. Note that the resulting FSM is a *non-deterministic finite state machine* (NFSM).

Assume we have an NFSM as above. Then (while ignoring groupings) the state of the ADT is a state of the NFSM and the operations of the ADT can easily be mapped to the FSM. Testing for a logical ordering can be performed by checking if the node with the ordering is reachable from the current state by following ϵ edges. If the set must be changed because of a functional de-

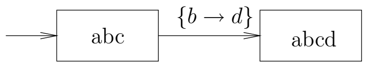

Figure 8.3.: Possible FSM for groupings

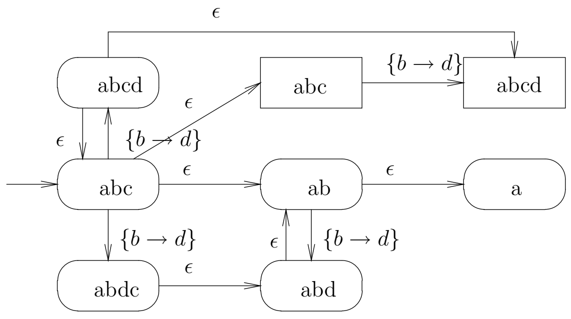

Figure 8.4.: Combined FSM for orderings and groupings

pendency the state is changed by following the edge labeled with the functional dependency. Of course, the non-determinism stands in our way.

While remembering only the active state of the NFSM avoids the problem of maintaining a set of orderings, the NFSM is not really useful from a practical point of view, since the transitions are non-deterministic. Nevertheless, the NFSM can be considered as a special *non-deterministic finite automaton* (NFA), which consumes the functional dependencies and "recognizes" the possible physical orderings. Further, an NFA can be converted into a *deterministic finite automaton* (DFA), which can be handled efficiently. Remember that the construction is based on the power set of the NFA's states. That is, the states of the DFA are sets of states of the NFA [47]. We do not take the deviation over the finite automaton but instead lift the construction of deterministic finite automatons from non-deterministic ones to finite state machines. Since this is not a traditional conversion, we give a proof of this step in Section 8.6.

Yet another problem is that the conversion from an NFSM to a *deterministic FSM* (DFSM) can be expensive for large NFSMs. Therefore, reducing the size of the NFSM is another problem we look at. We introduce techniques for reducing the set of functional dependencies that have to be considered and further techniques to prune the NFSM in Section 8.5.7.

The idea of a combined framework for orderings and groupings was presented in [61]. Here, the main point is to construct a similar FSM for groupings and integrate it into the FSM for orderings, thus handling orderings and groupings at the same time. An example of this is shown in Figure 8.3. Here, the FSM for the grouping $\{a, b, c\}$ and the functional dependency $b \to c$ is shown. We represent states for orderings as rounded boxes and states for groupings as rectangles.

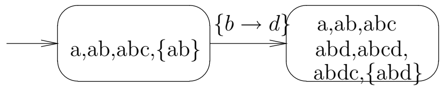

Figure 8.5.: Possible DFSM for Figure 8.4

Note that although the FSM for groupings has a start node similar to the FSM for orderings, it is much smaller. This is due to the fact that groupings are only compatible with themselves, no nodes for prefixes are required. However, the FSM is still non-deterministic: given the functional dependency $b \rightarrow c$, the grouping $\{a, b, c, d\}$ is compatible with $\{a, b, c, d\}$ itself and with $\{a, b, c\}$; therefore, there exists an (implicit) edge from each grouping to itself.

The FSM for groupings is integrated into the FSM for orderings by adding ϵ edges from each ordering to the grouping with the same attributes; this is due to the fact that every ordering is also a grouping. Note that although the ordering (a, b, c, d) also implies the grouping $\{a, b, c\}$, no edge is required for this, since there exists an ϵ edge to (a, b, c) and from there to $\{a, b, c\}$.

After constructing a combined FSM as described above, the full ADT supporting both orderings and groupings can easily be mapped to the FSM: The state of the ADT is a state of the FSM and testing for a logical ordering or grouping can be performed by checking if the node with the ordering or grouping is reachable from the current state by following ϵ edges (as we will see, this can be precomputed to yield the O(1) time bound for the ADT operations). If the state of the ADT must be changed because of functional dependencies, the state in the FSM is changed by following the edge labeled with the functional dependency.

However, the non-determinism of this transition is a problem. Therefore, for practical purposes the NFSM must be converted into a DFSM. The resulting DFSM is shown in Figure 8.5. Note that although in this simple example the DFSM is very small, the conversion could lead to exponential growth. Therefore, additional pruning techniques for groupings are presented in Section 8.5.7. However, the inclusion of groupings is not critical for the conversion, as the grouping part of the NFSM is nearly independent of the ordering part. In Section 8.7 we look at the size increase due to groupings. The memory consumption usually increases by a factor of two, which is the minimum expected increase, since every ordering is a grouping.

Some operators, like *sort*, change the physical ordering. In the NFSM, this is handled by changing the state to the node corresponding to the new physical ordering. Implied by its construction, in the DFSM this new physical ordering typically occurs in several nodes. For example, (a, b, c) occurs in both nodes of the DFSM in Figure 8.5. It is, therefore, not obvious which node to choose. We will take care of this problem during the construction of the NFSM (see Section 8.5.3).

8.5. Detailed Algorithm

8.5.1. Overview

Our approach consists of two phases. The first phase is the preparation step taking place before the actual plan generation starts. The output of this phase are the precomputed values used to implement the ADT. Then the ADT is used during the second phase where the actual plan generation takes place. The first

1. Determine the input
    a) Determine interesting orders
    b) Determine interesting groupings
    c) Determine set of functional dependencies
2. Construct the NFSM
    a) Construct states of the NFSM
    b) Filter functional dependencies
    c) Build filters for orderings and groupings
    d) Add edges to the NFSM
    e) Prune the NFSM
    f) Add artificial start state and edges
3. Convert the NFSM into a DFSM
4. Precompute values
    a) Precompute the compatibility matrix
    b) Precompute the transition table

Figure 8.6.: Preparation steps of the algorithm

phase is performed exactly once and is quite involved. Most of this section covers the first phase. Only Section 8.5.6 deals with the ADT implementation.

Figure 8.6 gives an overview of the preparation phase. It is divided into four major steps, which are discussed in the following subsections. Subsection 8.5.2 briefly reviews how the input to the first phase is determined and, more importantly, what it looks like. Section 8.5.3 describes in detail the construction of the NFSM from the input. The conversion from the NFSM to the DFSM is only briefly sketched in Section 8.5.4, for details see [47]. From the DFSM some values are precomputed which are then used for the efficient implementation of the ADT. The precomputation is described in Section 8.5.5, while their utilization and the ADT implementation are the topic of Section 8.5.6. Section 8.5.7 contains some important techniques to reduce the size of the NFSM. They are applied in Steps 2 (b), 2 (c) and 2 (e). During the discussion, we illustrate the different steps by a simple running example. More complex examples can be found in Section 8.7.

8.5.2. Determining the Input

Since the preparation step is performed immediately before plan generation, it is assumed that the query optimizer has already determined which indices are applicable and which algebraic operators can possibly be used to construct the query execution plan.

Before constructing the NFSM, the set of interesting orders, the set of interesting groupings and the sets of functional dependencies for each algebraic operator are determined. We denote the set of sets of functional dependencies by $\mathcal{F}$. It is important for the correctness of our algorithms that we note which of the interesting orders are (1) produced by some algebraic operator or (2) only tested for. Note that the interesting orders which satisfy (1) may additionally be tested for as well. We denote those orderings under (1) by O_P, those under (2) by O_T. The total set of interesting orders is defined as $O_I = O_P \cup O_T$. The orders produced are treated slightly differently in the following steps. For details on determining the set of interesting orders we refer to [69, 70]. The groupings are classified similarly to the orderings: We denote the grouping produced by some algebraic operator by G_P, and those just tested for by G_T. The total set of interesting groupings is defined as $G_I = G_P \cup G_T$. More information on how to extract interesting groupings can be found in [77]. Furthermore, for a sample query the extraction of both interesting orders and groupings is illustrated in Section 8.7.

To illustrate subsequent steps, we assume that the set of sets of functional dependencies

$$\mathcal{F} = \{\{b \to c\}, \{b \to d\}\},$$

the interesting groupings

$$G_I = \{\{b\}\} \cup \{\{b, c\}\}$$

and the interesting orders

$$O_I = \{(b), (a, b)\} \cup \{(a, b, c)\}$$

have been extracted from the query. We assume that those in $O_T = \{(a, b, c)\}$ and $G_T = \{\{b, c\}\}$ are tested for but not produced by any operator, whereas those in $O_P = \{(b), (a, b)\}$ and $G_P = \{\{b\}\}$ may be produced by some algebraic operators.

8.5.3. Constructing the NFSM

An NFSM consists of a tuple (Σ, Q, D, q_o), where

- Σ is the input alphabet,
- Q is the set of possible states,
- $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, and
- q_0 is the initial state.

Coarsely, Σ consists of the functional dependencies, Q of the relevant orderings and groupings, and D describes how the orderings or groupings change under a given functional dependency. Some refinements are needed to provide efficient ADT operations. The details of the construction are described now.

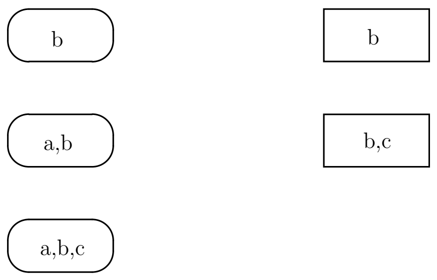

Figure 8.7.: Initial NFSM for sample query

For the order optimization part the states are partitioned in $Q = Q_I \cup Q_A \cup \{q_0\}$, where q_0 is an artificial state to initialize the ADT, Q_I is the set of states corresponding to interesting orderings and Q_A is a set of artificial states only required for the algorithm itself. Q_A is described later. Furthermore, the set Q_I is partitioned in Q_I^P and Q_I^T, representing the orderings in O_P and O_T, respectively. To support groupings, we add to Q_I^P states corresponding to the groupings in G_P and to Q_I^T states corresponding to the groupings in G_T.

The initial NFSM contains the states Q_I of interesting groupings and orderings. For the example, this initial construction not including the start state q_o is shown in Figure 8.7. The states representing groupings are drawn as rectangles and the states representing orderings are drawn with rounded corners.

When considering functional dependencies, additional groupings and orderings can occur. These are not directly relevant for the query, but have to be represented by states to handle transitive changes. Since they have no direct connection to the query, these states are called artificial states. Starting with the initial states Q_I, artificial states are constructed by considering functional dependencies

$$Q_A = (\Omega(O_I, \mathcal{F}) \setminus O_I) \cup (\Omega(G_I, \mathcal{F}) \setminus G_I).$$

In our example, this creates the states (b, c) and (a), as (b, c) can be inferred from (b) when considering $\{b \rightarrow c\}$ and (a) can be inferred from (a, b), since (a) is a prefix of (a, b). The result is show in Figure 8.8 (ignore the edges).

Sometimes the ADT has to be explicitly initialized with a certain ordering or grouping (e.g. after a `sort`). To support this, artificial edges are added later on. These point to the requested ordering or grouping (states in Q_I^P) and are labeled with the state that they lead to. Therefore, the input alphabet Σ consists of the sets of functional dependencies and produced orderings and groupings:

$$\Sigma = \mathcal{F} \cup Q_I^P \cup \{\epsilon\}.$$

In our example, $\Sigma = \{\{b \rightarrow c\}, \{b \rightarrow d\}, (b), (a, b), \{b\}\}$.

Accordingly, the domain of the transition relation D is

$$\begin{aligned} D \subseteq & \quad ((Q \setminus \{q_0\}) \times (\mathcal{F} \cup \{\epsilon\}) \times (Q \setminus \{q_0\})) \\ \cup & \quad (\{q_o\} \times Q_I^P \times Q_I^P). \end{aligned}$$

The edges are formed by the functional dependencies and the artificial edges. Furthermore, ϵ edges exist between orderings and the corresponding groupings,

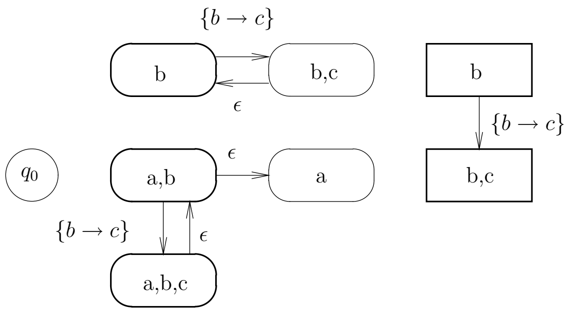

Figure 8.8.: NFSM after adding D_{FD} edges

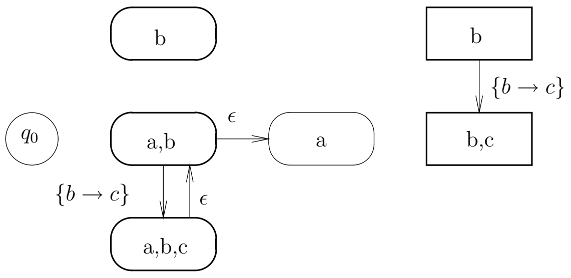

Figure 8.9.: NFSM after pruning artificial states

as orderings are a special case of grouping:

$$\begin{aligned}
D_{FD} &= \{(q, f, q') \mid q \in Q, f \in \mathcal{F} \cup \{\epsilon\}, q' \in Q, q \vdash_f q'\} \\
D_A &= \{(q_0, q, q) \mid q \in Q_I^P\} \\
D_{OG} &= \{(o, \epsilon, g) \mid o \in \Omega(O_I, \mathcal{F}), g \in \Omega(G_I, \mathcal{F}), o \equiv g\} \\
D &= D_{FD} \cup D_A \cup D_{OG}
\end{aligned}$$

First, the edges corresponding to functional dependencies are added (D_{FD}). In our example, this results in the NFSM shown in Figure 8.8.

Note that the functional dependency $b \to d$ has been pruned, since d does not occur in any interesting order or grouping. The NFSM can be further simplified by pruning the artificial state (b, c), which cannot lead to a new interesting order. The result is shown in Figure 8.9. A detailed description of these pruning techniques can be found in Section 8.5.7.

The artificial start state q_0 has emanating edges incident to all states representing interesting orders in O_I^P and interesting groupings in G_I^P (D_A). Also, the states representing orderings have edges to their corresponding grouping states (D_{OG}), as every ordering is also a grouping. The final NFSM for the example is shown in Figure 8.10. Note that the states representing (a, b, c) and $\{b, c\}$ are not linked by an artificial edge since it is only tested for, as they are in Q_I^T.

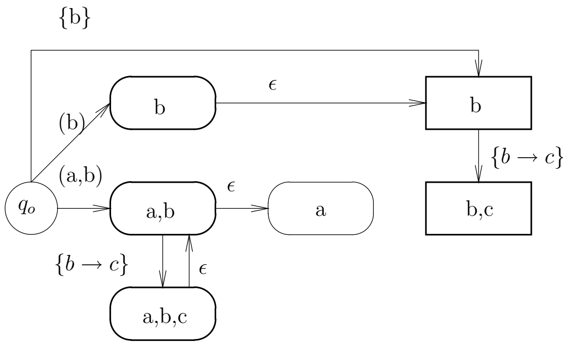

Figure 8.10.: Final NFSM

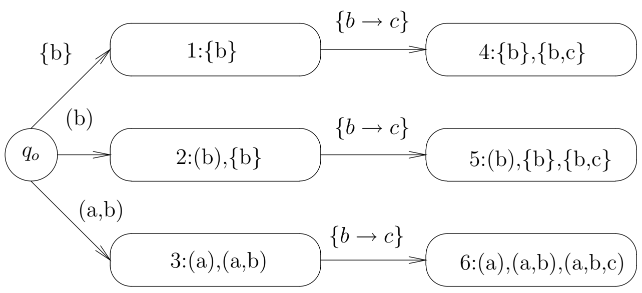

Figure 8.11.: Resulting DFSM

8.5.4. Constructing the DFSM

The construction of the DFSM from the NFSM follows the standard power set construction that is used to translate an NFA into a DFA [47]. A formal description and a proof of correctness is given in Section 8.6. It is important to note that this construction preserves the start state and the artificial edges. The resulting DFSM for the example is shown in Figure 8.11.

8.5.5. Precomputing Values

To allow for an efficient precomputation of values, every occurrence of an interesting order, interesting grouping or set of functional dependencies is replaced

state	1: (a)	2: (a,b)	3: (a,b,c)	4: (b)	5: {b}	6: {b,c}
1	0	0	0	0	1	0
2	0	0	0	1	1	0
3	1	1	0	0	0	0
4	0	0	0	0	1	1
5	0	0	0	1	1	1
6	1	1	1	0	0	0

Figure 8.12.: *contains* Matrix

state	1: $\{b \rightarrow c\}$	2: (a, b)	3: (b)	4: $\{b\}$
q_o	-	3	2	1
1	4	-	-	-
2	5	-	-	-
3	6	-	-	-
4	4	-	-	-
5	5	-	-	-
6	6	-	-	-

Figure 8.13.: *transition* Matrix

by integers. This allows comparisons in constant time (equivalent entries are mapped to same integer). Further, the DFSM is represented by an adjacency matrix.

The precomputation step itself computes two matrices. The first matrix denotes whether an NFSM state in Q_I is active, i.e. an interesting order or an interesting grouping, is contained in a specific DFSM state. This matrix can be represented as a compact bit vector, allowing tests in O(1). For our running example, it is given (in a more readable form) in Figure 8.12. The second matrix contains the transition table for the DFSM relation D. Using it, edges in the DFSM can be followed in $O(1)$. For the example, the transition matrix is given in Figure 8.13.

8.5.6. During Plan Generation

During plan generation, larger plans are constructed by adding algebraic operators to existing (sub-)plans. Each subplan contains the available orderings and groupings in the form of the corresponding DFSM state. Hence, the state of the DFSM, a simple integer, is the state of our ADT `OrderingGrouping`.

When applying an operator to subplans, the ordering and grouping requirements are tested by checking whether the DFSM state of the subplan contains the required ordering or grouping of the operator. This is done by a simple lookup in the *contains* matrix.

If the operator introduces a new set of functional dependencies, the new state of the ADT is computed by following the according edge in the DFSM. This is performed by a quick lookup in the *transition* matrix.

For “atomic” subplans like table or index scans, the ordering and grouping is determined explicitly by the operator. The state of the DFSM is determined by a lookup in the transition matrix with start state q_o and the edge annotated by the produced ordering or grouping. For sort and group-by operators the state of the DFSM is determined as before by following the artificial edge for the produced ordering or grouping and then reapplying the set of functional dependencies that currently hold.

In the example, a sort on (b) results in a subplan with ordering/grouping state 2 (the state 2 is active in the DFSM), which satisfies the ordering (b)

and the grouping $\{b\}$. After applying an operator which induces $b \to c$, the ordering/grouping changes to state 5 which also satisfies $\{b, c\}$.

8.5.7. Reducing the Size of the NFSM

Reducing the size of the NFSM is important for two reasons: First, it reduces the amount of work needed during the preparation step, especially the conversion from NFSM to DFSM. Even more important is that a reduced NFSM results in a smaller DFSM. This is crucial for plan generation, since it reduces the search space: Plans can only be compared and pruned if they have comparable ordering and a comparable set of functional dependencies (see [70] for details). Reducing the size of the DFSM removes information that is not relevant for plan generation and, therefore, allows a more aggressive pruning of plans.

At first, the functional dependencies are pruned. Here, functional dependencies which can never lead to a new interesting order or grouping are removed. For convenience, we extend the definition of $\Omega(O, F)$ and define

$$\Omega(O, \epsilon) \;:=\; \Omega(O, \emptyset).$$

Then the set of prunable functional dependencies F_P can be described by

$$\begin{aligned} \Omega_N(o, f) &:= \Omega(\{o\}, \{f\}) \setminus \Omega(\{o\}, \epsilon) \\ F_P &:= \{f | f \in F \wedge \forall o \in O_I \cup G_I : \\ &\quad\; (\Omega(\Omega_N(o, f), F) \setminus \Omega(\{o\}, \epsilon)) \cap (O_I \cup G_I) = \emptyset\}. \end{aligned}$$

Pruning functional dependencies is especially useful, since it also prunes artificial states that would be created because of the dependencies. In the example, this removed the functional dependency $b \to d$, since d does not appear in any interesting order or grouping. This step also removes the artificial states containing d.

The artificial states are required to build the NFSM, but they are not visible outside the NFSM. Therefore, they can be pruned and merged without affecting plan generation. Two heuristics are used to reduce the set of artificial states:

1. All artificial nodes that behave exactly the same (that is, their edges lead to the same states given the same input) are merged and

2. all edges to artificial states that can reach states in Q_I only through ϵ edges are replaced with corresponding edges to the states in Q_I.

More formally, the following pairs of states can be merged:

$$\begin{aligned} \{(o_1, o_2) \;\;|\;\; & o_1 \in Q_A, o_2 \in Q_A \wedge \forall f \in F : \\ & (\Omega(\{o_1\}, \{f\}) \setminus \Omega(\{o_1\}, \epsilon)) = \\ & (\Omega(\{o_2\}, \{f\}) \setminus \Omega(\{o_2\}, \epsilon))\}. \end{aligned}$$

The following states can be replaced with the next state reachable by an ϵ edge:

$$\begin{aligned}\{o \quad | \quad & o \in Q_A \wedge \forall f \in F : \\ & \Omega(\Omega(\{o\}, \epsilon), \{f\}) \setminus \{o\} = \\ & \Omega(\Omega(\{o\}, \epsilon) \setminus \{o\}, \{f\})\}.\end{aligned}$$

In the example, this removed the state (b, c), which was artificial and only led to the state (b).

These techniques reduce the size of the NFSM, but still most states are artificial states, i.e. they are only created because they can be reached by considering functional dependencies when a certain ordering or grouping is available. But many of these states are not relevant for the actual query processing. For example, given a set of interesting orders which consists only of a single ordering (a) and a set of functional dependencies which consists only of $a \rightarrow b$, the NFSM will contain (among others) two states: (a) and (a, b). The state (a, b) is created since it can be reached from (a) by considering the functional dependency, however, it is irrelevant for the plan generation, since (a, b) is not an interesting order and is never created nor tested for. Actually, in the example above, the whole functional dependency would be pruned (since b never occurs in an interesting order), but the problem remains for combinations of interesting orders: Given the interesting orders (a), (b) and (c) and the functional dependencies $\{a \rightarrow b, b \rightarrow a, b \rightarrow c, c \rightarrow b\}$, the NFSM will contain states for all permutations of a, b and c. But these states are completely useless, since all interesting orders consist only of a single attribute and, therefore, only the first entry of an ordering is ever tested.

Ideally, the NFSM should only contain states which are relevant for the query; since this is difficult to ensure, a heuristic can be used which greatly reduces the size of the NFSM and still guarantees that all relevant states are available: When considering a functional dependency of the form $a \rightarrow b$ and an ordering $o_1, o_2, \ldots, o_n$ with $o_i = a$ for some i $(1 \leq i \leq n)$, the b can be inserted at any position j with $i < j \leq n + 1$ (for the special case of a condition $a = b$, $i = j$ is also possible). So, an entry of an ordering can only affect entries on the right of its own position. This means that it is unnecessary to consider those parts of an ordering which are behind the length of the longest interesting order; since that part cannot influence any entries relevant for plan generation, it can be omitted. Therefore, the orderings created by functional dependencies can be cut off after the maximum length of interesting orders, which results in less possible combinations and a smaller NFSM.

The space of possible orderings can be limited further by taking into account the prefix of the ordering: before inserting an entry b in an ordering $o_1, o_2, \ldots, o_n$ at the position i, check if there is actually an interesting order with the prefix $o_1, o_2, ...o_{i-1}, b$ and stop inserting if no interesting order is found. Also limit the new ordering to the length of the longest matching interesting order; further attributes will never be used. If functional dependencies of the form $a = b$ occur, they might influence the prefix of the ordering and the simple test described above is not sufficient. Therefore, a representative is chosen for each equivalence class created by these dependencies, and for the prefix test the attributes are replaced with their representatives. Since the set of interesting orders with a prefix of $o_1, \ldots, o_n$ is a superset of the set for the prefix $o_1, ...o_n, o_{n+1}$, this heuristic can be implemented very efficiently by iterating over i and reducing the set as needed.

Additional techniques can be used to avoid creating superfluous artifical states for groupings: First, in Step 2.3 (see Figure 8.6) the set of attributes occurring in interesting groupings is determined:

$$A_G = \{a \mid \exists g \in G_I : a \in g\}$$

Now, for every attribute a occurring on the right-hand side of a functional dependency the set of potentially reachable relevant attributes is determined:

$$\begin{aligned}
r(a,0) &= \{a\} \\
r(a,n) &= r(a,n-1) \cup \\
&\quad \{a' \mid \exists(a_1 \ldots a_m \to a') \in \mathcal{F} : \\
&\qquad \{a_1 \ldots a_m\} \cap r(a,n-1) \neq \emptyset\} \\
r(a) &= r(a,|\mathcal{F}|) \cap A_G
\end{aligned}$$

This can be used to determine if a functional dependency actually adds useful attributes. Given a functional dependency $a_1 \ldots a_n \to a$ and a grouping g with $\{a_1 \ldots a_n\} \subseteq g$, a should only be added to g if $r(a) \not\subseteq g$, i.e. the attribute might actually lead to a new interesting grouping. For example, given the interesting groupings $\{a\}, \{a,b\}$ and the functional dependencies $a \to c, a \to d, d = b$. When considering the grouping $\{a\}$, the functional dependency $a \to c$ can be ignored, as it can only produce the attribute c, which does not occur in an interesting grouping. However, the functional dependency $a \to d$ should be added, since transitively the attribute b can be produced, which does occur in an interesting grouping.

Since there are no ϵ edges between groupings, i.e. groupings are not compatible with each other, a grouping can only be relevant for the query if it is a subset of an interesting ordering (as further attributes could be added by functional dependencies). However, a simple subset test is not sufficient, as equations of the form $a = b$ are also supported; these can effectively rename attributes, resulting in a slightly more complicated test:

In Step 2.3 (see Figure 8.6) the equivalence classes induced by the equations in $\mathcal{F}$ are determined and for each class a representative is chosen (a and $a_1 \ldots a_n$ are attributes occuring in the G_I):

$$\begin{aligned}
E(a,0) &= \{a\} \\
E(a,n) &= E(a,n-1) \cup \\
&\quad \{a' \mid ((a = a') \in \mathcal{F}) \vee ((a' = a) \in \mathcal{F})\} \\
E(a) &= E(a,|\mathcal{F}|) \\
e(a) &= \text{a representative choosen from } E(A) \\
e(\{a_1 \ldots a_n\}) &= \{e(a_1) \ldots e(a_n)\}.
\end{aligned}$$

Using these equivalence classes, a mapped set of interesting groupings is produced that will be used to test if a grouping is relevant:

$$G_I^E \;=\; \{e(g) \mid g \in G_I\}$$

Now a grouping g can be pruned if $\nexists g' \in G_I^E : e(g) \subseteq g'$. For example, given the interesting grouping $\{a\}$ and the equations $a = b, b = c$, the grouping $\{d\}$ can be pruned, as it will never lead to an interesting grouping; however, the groupings $\{b\}$ and $\{c\}$ have to be kept, as they could change to an interesting grouping later on.

Note that although they appear to test similar conditions, the first pruning technique (using $r(a)$) is not dominated by the second one (using $e(a)$). Consider e.g. the interesting grouping $\{a\}$, the equation $a = b$ and the functional dependency $a \rightarrow b$. Using only the second technique, the grouping $\{a, b\}$ would be created, although it is not relevant.

8.5.8. Complex Ordering Requirements

Specifying the ordering requirements of an operator can be surprisingly difficult. Consider the following SQL query:

```
select *
from   S s, R r
where  r.a=s.a and r.b=s.b and
       r.c=s.c and r.d=s.d
```

When answering this query using a sort-merge join, the operator has to request a certain odering. But there are many orderings that could be used: The intuitive ordering would be *abcd*, but *adcb* or any other premutation could have been used as well. This is problematic, as checking for an exponential number of possibilities is not acceptable in general. Note that this problem is not specific to our approach, the same is true, e.g., for Simmen's approach.

The problem can be solved by defining a total ordering between the attributes, such that a canonical ordering can be constructed. We give some rules how to derive such an ordering below, but it can happen that such an ordering is unavailable (or rather the construction rules are ambiguous). Given, for example, two indices, one on *abcd* and one on *adcb*, both orderings would be a reasonable choice. If this happens, the operators have two choices: Either they accept all reasonable orderings (which could still be an exponential number, but most likely only a few orderings remaing) or they limit themselves to one ordering, which could induce unnecessary sort operators. Probably the second choice is preferable, as the ambiguous case should be rare and does not justify the complex logic of the first solution.

The attribute ordering can be derived by using the following heuristical rules:

1. Only attributes that occur in sets without natural ordering (i.e. complex join predicates or grouping attributes) have to be ordered.

2. Orderings that are given (e.g., indices, user-requested orderings etc.) order some attributes.

3. Small orderings should be considered first. If an operator requires an ordering with the attributes *abc*, and another operator requires an ordering with the attributes *bc*, the attributes b and c should come before a.

4. The attributes should be ordered according to equivalence classes. If a is ordered before b, all orderings in $E(a)$ should be ordered before all orderings in $E(b)$.

5. Attributes should be ordered according to the functional dependencies, i.e. if $a \rightarrow b$, a should come before b. Note that $a = b$ suggests no ordering between a and b.

6. The remaining unordered attributes can be ordered in an arbitrary way.

The rules must check if they create contradictions. If this happens. the contradicting ordering must be omitted, resulting in potentially superfluous sort operators. Note that in some cases these sort operators are simply unavoidable: If for the example query one index on R exists with the ordering *abcd* and one index on S with the ordering *dcba*, the heuristical rules detect a contradiction and choose one of the orderings. This results in a sort operator before the (sort-merge) join, but this sort could not have been avoided anyway.

8.6. Converting a NFSM into a DFSM

The algorithm described in this chapter first constructs a non-deterministic FSM and converts it to a deterministic FSM. For this conversion, the NFSM is treated like an NFA which is converted to a DFA. It has to be shown that the DFSM resulting from the conversion is equivalent to the initial NFSM:

8.6.1. Definitions

An NFA [47] consists of a tuple (Σ, Q, D, q_o, F), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation, q_0 the initial state and F the set of accepting states. All nodes reachable from a given set of nodes Q by following ϵ edges can be described by

$$\begin{aligned}
\mathcal{E}_D^0(Q) &= Q \\
\mathcal{E}_D^i(Q) &= \{q' | \exists q \in \mathcal{E}_D^{i-1}(Q), (q, \epsilon, q') \in D\} \\
\mathcal{E}_D(Q) &= \bigcup_{i=0}^{\infty} \mathcal{E}_D^i(Q)
\end{aligned}$$

Then the NFA *accepts* an input $w = w_1 w_2 ... w_n \in \Sigma^*$ if $S_n \cap F \neq \emptyset$ where

$$\begin{aligned}
S_0 &= \mathcal{E}_D(q_o) \\
S_i &= \mathcal{E}_D(\{q' | \exists q \in S_{i-1} : (q, w_i, q') \in D\}).
\end{aligned}$$

Similarly, a DFA [47] consists of a tuple $(\Sigma, Q, \Delta, q_o, F)$ where

$$\begin{aligned} & \Delta \subseteq Q \times \Sigma \times Q \\ \wedge \quad & \forall a, b, c \in Q, d \in \Sigma : \\ & ((a, d, b) \in \Delta \wedge (a, d, c) \in \Delta) \Rightarrow b = c. \end{aligned}$$

So a DFA is an NFA which only allows non-ambiguous non-ϵ transitions. The definition of accepting is analogous to the definition for NFAs.

An NFSM is basically an NFA without accepting states. It consists of a tuple (Σ, Q, D, q_o), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation and q_0 the initial state. While an NFSM does not have any accepting states it is usually important to know which state is active after a given input, so in a way each state is accepting.

Likewise, a DFSM basically is a DFA without accepting states. It consists of a tuple (Σ, Q, Δ, q_o) where Σ, Q,Δ and q_o are analogous to the DFA. Again, while there is no set of accepting states, it is important to know which one is active after a given input.

8.6.2. The Transformation Algorithm

The commonly used algorithm to convert an NFA into a DFA (see [47]) can also be used to convert an NFSM into a DFSM. Since the accepting states are not required for the algorithm, the NFSM can be regarded as an NFA and converted into a "DFA", which is really a DFSM. The correctness of this transformation is shown in the next section.

The algorithm converts an NFSM (Σ, Q, D, q_o) in a DFSM $(\Sigma, Q', \Delta, q'_0)$ with $Q' \subseteq 2^Q$. It first constructs a start node $q'_0 = \mathcal{E}_D(\{q_0\})$ and then determines for all DFSM nodes q' all outgoing edges δ' by expanding all edges in the contained NFSM nodes:

$$\begin{aligned} \delta(q') \quad = \quad & \{(q', \sigma, q'_2 | \sigma \in \Sigma, q'_2 \neq \emptyset, \\ & q'_2 = \{\mathcal{E}_D(q_2) | (q, \sigma, q_2) \in D, q \in q'\}\}. \end{aligned}$$

This results in the DFSM $(\Sigma, Q', \Delta, q'_o)$ with

$$\begin{aligned} Q'_0 &= \{q'_0\} \\ Q'_i &= \textstyle\bigcup_{q' \in Q'_{i-1}} \{q'_2 | \exists \sigma \in \Sigma : (q', \sigma, q'_2) \in \delta(q')\} \\ Q' &= \textstyle\bigcup_{i=0}^{\infty} Q'_i \\ \Delta &= \textstyle\bigcup_{q' \in Q'} \delta(q). \end{aligned}$$

8.6.3. Correctness of the FSM Transformation

Proposition: Given an NFSM (Σ, Q, D, q_o), the DFSM $(\Sigma, Q' \subseteq 2^Q, \Delta, q'_0)$ constructed by using the transformation algorithm for NFA to DFA described in

n	#Edges	t (ms)	#Plans	t/plan	t (ms)	#Plans	t/plan	% t	% #Plans	%. t/plan
5	n-1	2	1541	1.29	1	1274	0.78	2.00	1.21	1.65
6	n-1	9	7692	1.17	2	5994	0.33	4.50	1.28	3.55
7	n-1	45	36195	1.24	12	26980	0.44	3.75	1.34	2.82
8	n-1	289	164192	1.76	74	116562	0.63	3.91	1.41	2.79
9	n-1	1741	734092	2.37	390	493594	0.79	4.46	1.49	3.00
10	n-1	11920	3284381	3.62	1984	2071035	0.95	6.01	1.59	3.81
5	n	4	3060	1.30	1	2051	0.48	4.00	1.49	2.71
6	n	21	14733	1.42	4	9213	0.43	5.25	1.60	3.30
7	n	98	64686	1.51	20	39734	0.50	4.90	1.63	3.02
8	n	583	272101	2.14	95	149451	0.63	6.14	1.82	3.40
9	n	4132	1204958	3.42	504	666087	0.75	8.20	1.81	4.56
10	n	26764	4928984	5.42	2024	2465646	0.82	13.22	2.00	6.61
5	n+1	12	5974	2.00	1	3016	0.33	12.00	1.98	6.06
6	n+1	69	26819	2.57	6	12759	0.47	11.50	2.10	5.47
7	n+1	370	119358	3.09	28	54121	0.51	13.21	2.21	6.06
8	n+1	2613	509895	5.12	145	208351	0.69	18.02	2.45	7.42
9	n+1	27765	2097842	13.23	631	827910	0.76	44.00	2.53	17.41
10	n+1	202832	7779662	26.07	3021	3400945	0.88	67.14	2.29	29.62

Figure 8.14.: Plan generation for different join graphs, Simmen's algorithm (left) vs. our algorithm (middle)

[47] behaves exactly like the NFSM, i.e.

$$\begin{aligned} 1) &\quad \forall w \in \Sigma^*, q \in Q, q_0 \xrightarrow{w} q \ \exists q' \in Q' : q'_0 \xrightarrow{w} q' \wedge q \in q' \\ 2) &\quad \forall w \in \Sigma^*, q'_a \in Q', q'_b \in Q', q_a \in q'_a, q_b \in q'_b : \\ &\quad (q_a \xrightarrow{w} q_b) \text{ iff } (q'_a \xrightarrow{w} q'_b) \end{aligned}$$

Proof: Proposition 1) trivially follows from the definition of the transformation algorithm, see the definition of δ' and Q' in Section 8.6.2.

The proof for proposition 2) can be derived from the proof in [47], Chapter 2.3: there, it is shown that for all $w \in \Sigma^*$, given a node q in the NFA and a node q' in the transformed DFA with $q \in q'$, a node f' in the DFA contains a node f in the NFA if and only if $q \xrightarrow{w} f$ and $q' \xrightarrow{w} f'$. Since the DFSM is constructed using the same algorithm, this results in proposition 2).

Therefore, the conversion algorithm used to convert an NFA into a DFA can be used to convert the NFSM describing the ordering transitions to a DFSM that behaves the same way as the NFSM.

8.7. Experimental Results

The framework described in this chapter solves two problems: First, it provides an efficient represententation for reasoning about orderings and second, it allows keeping track of orderings and groupings at the same time. Since these topics are treated separately in the related work, the experimental results are split in two sections: In Section 8.8 the framework is compared to another published framework while only considering orderings, and in Section 8.9 the influence of groupings is evaluated.

8.8. Total Impact

We now consider how order processing influences the time needed for plan generation. Therefore, we implemented both our algorithm and the algorithm proposed by Simmen et al. [70] and integrated them into a bottom-up plan generator based on [48].

To get a fair comparison, we tuned Simmen's algorithm as much as possible. The most important measure was to cache results in order to eliminate repeated calls to the very expensive *reduce* operation. Second, since Simmen's algorithm requires dynamic memory, we implemented a specially tailored memory management. This alone gave us a speed up by a factor of three. We further tuned the algorithm by thoroughly profiling it until no more improvements were possible. For each order optimization framework the plan generator was recompiled to allow for as many compiler optimizations as possible. We also carefully observed that in all cases both order optimization algorithms produced the same optimal plan.

We first measured the plan generation times and memory usage for TPC-R Query 8. A detailed discussion of this query follows in Section 8.9, here we ignored the grouping properties to compare it with Simmen's algorithm. The result of this experiment is summarized in the following table. Since order optimization is tightly integrated with plan generation, it is impossible to exactly measure the time spent just for order optimization during plan generation. Hence, we decided to measure the impact of order optimization on the total plan generation time. This has the advantage that we can also (for the first time) measure the impact order optimization has on plan generation time. This is important since one could argue that we are optimizing a problem with no significant impact on plan generation time, hence solving a non-problem. As we will see, this is definitely not the case.

In subsequent tables, we denote by *t(ms)* the total execution time for plan generation measured in milliseconds, by *#Plans* the total number of subplans generated, by *t/plan* the average time (in microseconds) needed to introduce one plan operator, i.e. the time to produce a single subplan, and by *Memory* the total memory (in KB) consumed by the order optimization algorithms.

	Simmen	Our algorithm
t (ms)	262	52
#Plans	200536	123954
t/plan (μs)	1.31	0.42
Memory (KB)	329	136

From these numbers, it becomes obvious that order optimization has a significant influence on total plan generation time. It may come as a surprise that fewer plans need to be generated by our approach. This is due to the fact that the (reduced) FSM only contains the information relevant to the query, resulting in fewer states. With Simmen's approach, the plan generator can only discard plans if the ordering is the same and the set of functional dependencies is equal (respectively a subset). It does not recognize that the additional information is not relevant for the query.

In order to show the influence of the query on the possible gains of our algorithm, we generated queries with 5-10 relations and a varying number of join predicates —that is, edges in the join graph. We always started from a chain query and then randomly added some edges. For small queries we

n	#Edges	Simmen	Our Algorithm	DFSM
5	n-1	14	10	2
6	n-1	44	28	2
7	n-1	123	77	2
8	n-1	383	241	3
9	n-1	1092	668	3
10	n-1	3307	1972	4
5	n	27	12	2
6	n	68	36	2
7	n	238	98	3
8	n	688	317	3
9	n	1854	855	4
10	n	5294	2266	4
5	n+1	53	15	2
6	n+1	146	49	3
7	n+1	404	118	3
8	n+1	1247	346	4
9	n+1	2641	1051	4
10	n+1	8736	3003	5

Figure 8.15.: Memory consumption in KB for Figure 8.14

averaged the results of 100 queries and averaged 10 queries for large queries. The results of the experiment can be found in Fig. 8.14. In the second column, we denote the number of edges in terms of the number of relations (n) given in the first column. The next six columns contain (1) the total time needed for plan generation (in ms), (2) the number of (sub-) plans generated, and (3) the time needed to generate a subplan (in μs), i.e. to add a single plan operator, for (a) Simmen's algorithm (columns 3-5) and our algorithm (columns 6-8). The total plan generation time includes building the DFSM when our algorithm is used. The last three columns contain the improvement factors for these three measures achieved by our algorithm. More specifically, column *%* x contains the result of dividing the x column of Simmen's algorithm by the corresponding x column entry of our algorithm.

Note that we are able to keep the plan generation time below one second in most cases and three seconds in the worst case, whereas when Simmen's algorithm is applied, plan generation time can be as high as 200 seconds. This observation leads to two important conclusions:

1. Order optimization has a significant impact on total plan generation time.

2. By using our algorithm, significant performance gains are possible.

For completeness, we also give the memory consumption during plan generation for the two order optimization algorithms (see Fig. 8.15). For our approach, we also give the sizes of the DFSM which are included in the total memory consumption. All memory sizes are in KB. As one can see, our approach consumes about half as much memory as Simmen's algorithm.

8.9. Influence of Groupings

Integrating groupings in the order optimization framework allows the plan generator to easily exploit groupings and, thus, produce better plans. However, order optimization itself might become prohibitively expensive by considering groupings. Therefore, we evaluated the costs of including groupings for different queries.

Since adding support for groupings has no effect on the runtime behavior of the plan generator (all operations are still one table lookup), we measured the runtime and the memory consumption of the preparation step both with and without considering groupings. When considering groupings, we treated each interesting ordering also as an interesting grouping, i.e. we assumed that a grouping-based (e.g. hash-based) operator was always available as an alternative. Since this is the worst-case scenario, it should give an upper bound for the additional costs. All experiments were performed on a 2.4 GHz Pentium IV, using the gcc 3.3.1.

To examine the impact for real queries, we choose a more complex query from the well-known TPC-R benchmark ([75], Query 8):

```
select
   o_year,
   sum(case when nation = '[NATION]'
      then volume
      else 0
   end) / sum(volume) as mkt_share
from
   (select
      extract(year from o_orderdate) as o_year,
      l_extendedprice * (1-l_discount) as volume,
      n2.n_name as nation
   from  part,supplier,lineitem,orders,customer,
         nation n1,nation n2,region
   where
      p_partkey = l_partkey and
      s_suppkey = l_suppkey and
      l_orderkey = o_orderkey and
      o_custkey = c_custkey and
      c_nationkey = n1.n_nationkey and
      n1.n_regionkey = r_regionkey and
      r_name = '[REGION]' and
      s_nationkey = n2.n_nationkey and
      o_orderdate between date '1995-01-01' and
         date '1996-12-31' and
      p_type = '[TYPE]'
   ) as all_nations
   group by o_year
   order by o_year;
```

When considering this query, all attributes used in joins, group-by and order-by clauses are added to the set of interesting orders. Since hash-based solutions are possible, they are also added to the set of interesting groupings. This results in the sets

$$\begin{aligned}
O_I^P &= \{(o\_year), (o\_partkey), (p\_partkey),\\
&\quad (l\_partkey), (l\_suppkey), (l\_orderkey),\\
&\quad (o\_orderkey), (o\_custkey), (c\_custkey),\\
&\quad (c\_nationkey), (n1.n\_nationkey),\\
&\quad (n2.n\_nationkey), (n\_regionkey),\\
&\quad (r\_regionkey), (s\_suppkey), (s\_nationkey)\}\\
O_I^T &= \emptyset\\
G_I^P &= \{\{o\_year\}, \{o\_partkey\}, \{p\_partkey\},\\
&\quad \{l\_partkey\}, \{l\_suppkey\}, \{l\_orderkey\},\\
&\quad \{o\_orderkey\}, \{o\_custkey\}, \{c\_custkey\},\\
&\quad \{c\_nationkey\}, \{n1.n\_nationkey\},\\
&\quad \{n2.n\_nationkey\}, \{n\_regionkey\},\\
&\quad \{r\_regionkey\}, \{s\_suppkey\}, \{s\_nationkey\}\}\\
G_I^T &= \emptyset
\end{aligned}$$

Note that here O_I^T and G_I^T are empty, as we assumed that each ordering and grouping would be produced if beneficial. For example, we might assume that it makes no sense to intentionally group by *o_year*: If a tuple stream is already grouped by *o_year* it makes sense to exploit this, however, instead of just grouping by *o_year* it could make sense to sort by *o_year*, as this is required anyway (although here it only makes sense if the sort operator performs early aggregation). In this case, $\{o\_year\}$ would move from G_I^P to G_I^T, as it would be only tested for, but not produced.

The set of functional dependencies (and equations) contains all join conditions and constant conditions:

$$\begin{aligned}
\mathcal{F} &= \{\{p\_partkey = l\_partkey\}, \{\emptyset \rightarrow p\_type\},\\
&\quad \{o\_custkey = c\_custkey\}, \{\emptyset \rightarrow r\_name\},\\
&\quad \{c\_nationkey = n1.n\_nationkey\},\\
&\quad \{s\_nationkey = n2.n\_nationkey\},\\
&\quad \{l\_orderkey = o\_orderkey\},\\
&\quad \{s\_suppkey = l\_suppkey\},\\
&\quad \{n1.n\_regionkey = r\_regionkey\}\}
\end{aligned}$$

To measure the influence of groupings, the preparation step was executed twice: Once with the data as given above and once with $G_I^P = \emptyset$ (i.e. groupings were ignored). The space and time requirements are shown below:

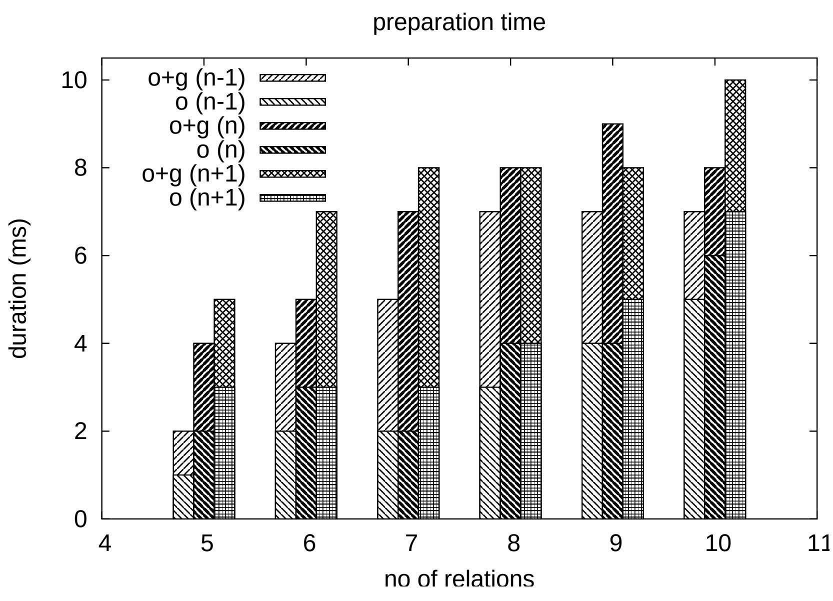

Figure 8.16.: Time requirements for the preparation step

	With Groups	Without Groups
Duration [ms]	0.6ms	0.3ms
DFSM [nodes]	63	32
Memory [KB]	5	2

Here time and space requirements both increase by a factor of two. Since all interesting orderings are also treated as interesting groupings, a factor of about two was expected.

While Query 8 is one of the more complex TPC-R queries, it is not overly complex when looking at order optimization. It contains 16 interesting orderings/groupings and 8 functional dependencies, but they cannot be combined in many reasonable ways, resulting in a comparatively small DFSM. In order to get more complex examples, we produced randomized queries with $5 - 10$ relations and a varying number of join predicates. We always started from a chain query and then randomly added additional edges to the join graph. The results are shown for $n - 1$, n and $n + 1$ additional edges. In the case of 10 relations, this means that the join graph consisted of 18, 19 and 20 edges, respectively.

The time and space requirements for the preparation step are shown in Figure 8.16 and Figure 8.17, respectively. For each number of relations, the requirements for the combined framework (`o+g`) and the framework ignoring groupings (`o`) are shown. The numbers in parentheses ($n-1$, n and $n+1$) are the number of additional edges in the join graph.

As with Query 8, the time and space requirements roughly increase by a factor of two when adding groupings. This is a very positive result, given that a factor of two can be estimated as a lower bound (since every interesting ordering is also an interesting grouping here). Furthermore, the absolute time and space requirements are very low (a few ms and a few KB), encouraging the inclusion of groupings in the order optimization framework.

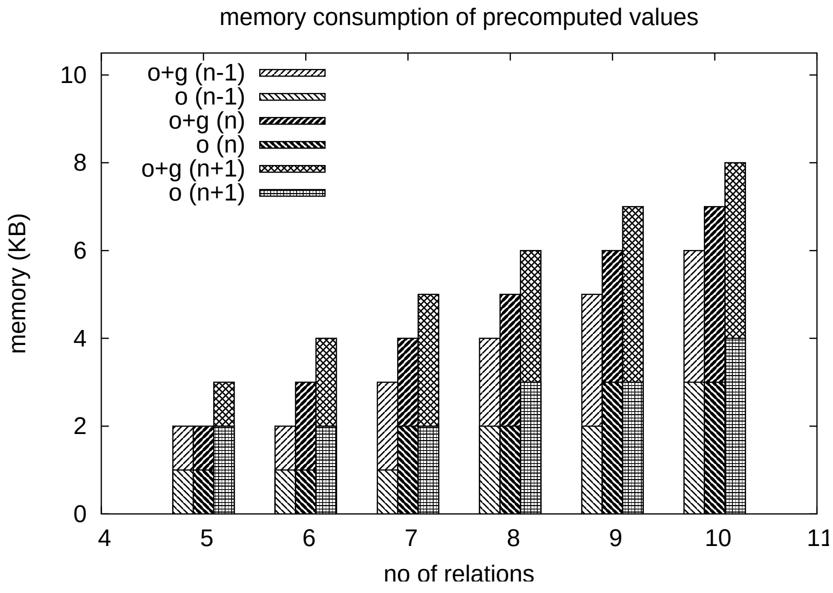

Figure 8.17.: Space requirements for the preparation step

8.10. Conclusion

The framework presented in this chapter allows a very efficient handling of order optimization during plan generation. After a preparation step with reasonable performance, the plan generation can change and test for orderings in O(1), using only O(1) space per subplan. Experimental results have shown that this can significantly reduce the time needed for plan generation by both reducing the time needed per subplan and the search space, which is essential for handling large queries.

Furthermore, the experimental results showed that with only a modest increase of the one-time costs, groupings can be exploited during plan generation at no additional costs. In summary, using an FSM to keep track of the available orderings and groupings is very efficient and is easily integrated in a plan generator.

One topic for future work is the minimization of the DFSM using the operator structure. Currently, only the NFSM is pruned by detecting irrelevant or redundant nodes. The DFSM could also be pruned by intentionally dropping available logical orderings or groupings when it is clear that the ordering or grouping will never be used (because of operator dependencies). Besides minimizing the DFSM, this technique would also reduce the search space for the plan generator, as more plans could be pruned (since more plans would be dominated by other plans).

9. Cost Model

9.1. Introduction

In order to find the optimal plan for a given query, the query optimizer has to decide if a given plan is better than another plan. This is usually done by choosing the "cheapest" plan, which assumes that the costs of a plan can be computed. Since the optimizer bases its decisions only on these costs, it is important that the costs are computed correctly, at least in the sense that a better plan indeed has the lower costs. Note that the notion of a better plan is actually ambiguous: In this chapter, we usually assume that we want to minimize the total costs of a query. However, sometimes it makes sense to minimize the costs for the first result tuple or the resource consumption during query execution. But regardless of the actual goals, the query optimizer needs a way to calculate the (relative) costs of a plan. This is provided by a separate program module, the cost model.

The first query optimizers only considered scans and joins and simply assumed that the costs of an operation are proportional to the number of tuples involved [79]. While this is sufficient to avoid the worst plans, it is only a very rough estimate of the actual costs, as it completely ignores the actual implementations of, e.g., join operators. Therefore, query optimizers soon tried to describe the real costs of a plan by estimating the time it would take to execute the plan [69]. This was usually done by calculating a weighted sum of expected costs for I/O and CPU [49].

This approach is much more accurate than just counting tuples, however, estimating the CPU costs and especially the I/O costs is not easy. Approximations for the number of accessed pages have been made quite early [82] and the costs for different operators have been estimated [30]. However these concentrate on joins and sorting, estimates for more complex operators like group-by are still incomplete [34].

Besides being somewhat inaccurate, the existing cost models are not directly applicable for DAG-structured query graphs. As they do not take into account that the output of an operator can be shared by multiple other operators, they severely overestimate the costs for DAGs. In this chapter, we present a framework that can be integrated into the plan generator to compare plans and to accurately keep track of the costs even if the query graph is a DAG.

The rest of this chapter is structured as follows: In Section 9.2 we describe the related work, especially concerning DAGs. Section 9.3 contains the interface provided for the plan generator and Section 9.4 sketches a concrete implementation of the interface. Section 9.5 describes the algorithm to handle DAGs and Section 9.6 discusses the actual impact of the cost model. Conclusions are drawn in Section 9.7.

9.2. Related Work

The general development of cost models and the corresponding papers were already discussed in Section 9.1. Besides, an abundance of related papers exists that usually concentrate on a very specific aspect of cost models and the related statistics. We will discuss some representatives below.

Few papers provide an overview of cost functions for all (or at least the popular) operators required for query optimizations. An overview for join costs is included in [72], but it neglects group-by etc. A discussion of a much wider range of operators can be found in [22], but this is more a benchmark with calculations than a full cost model (although the formulas could be used for one).

Some papers look at the physical properties of the used hardware to make cost models more exact. In [30] a detailed cost model for joins is described. The authors emphasize that the cost model should distinguish between random and sequential reads, as the cost differences are very large when using modern hard disks. A detailed description of the characteristics of CPUs and disks can be found in [65], but is only used for hybrid hash joins.

Other papers look more at the logical properties relevant for the cost model. Nearly all cost functions require the correct input cardinality to give reasonable results. However, only the cardinality of base relations can be directly derived from database statistics. The size of intermediate results is discussed in [19]. Another paper also looks at this and takes the concrete predicates into account [74]. Finally, [32] provides a detailed analysis of the number of tuples passed, the number of passes required (e.g. for sort) etc., but uses a very simplified disk model.

Other papers try to improve the precision of the predictions of the cost model. One way to do this is to use histograms for a more precise data distribution model [37]. A different approach is to accept the inaccuracy of the models and propagate error margins instead to get a more realistic view of the costs [36].

While cost models for query graphs with parallel execution exist [17], they do not specifically handle DAGs. The main problem there is that the operators are executed independently, and it is not clear which operators form the critical path and thus determine the total execution time.

9.3. Interface for Plan Generation

Before designing a concrete cost model, it is worthwhile to think about the interface between the plan generation and the cost model. While at first glance the interface seems to be obvious, some care is needed to allow more advanced cost models.

Note that while the interface shown below describes the logical interface of the cost model, it is still highly simplified. For real implementations a lot of additional code is required.

9.3.1. Goals

Before discussing the interface, we summarize the properties we would like the interface to have:

1. allow the plan generator to determine the optimal plan
2. allow arbitrary cost models
3. allow arbitrary operators
4. loose coupling between the plan generator, cost model and operators
5. minimize the overhead

While again these goals are somewhat obvious, they are difficult to accomplish at the same time, especially goals 2-4 contradict each other. In the following sections we sketch a compromise that tries to fulfill these goals with only a slight tighter coupling than preferable.

9.3.2. Cost Description

The most prominent part of the cost model is the cost description, determining if one plan is cheaper than the other. Historically, this has been a simple number, either the number of tuples involved [79] or some value proportional to the estimated execution time [69]. However, this cost description could be more complex, e.g., a vector of different properties like random reads, sequential reads etc. [65].

The plan generator itself does not need to know the structure of the cost description, for it it is sufficient to decide if one plan is better than the other. Therefore, each (partial) plan should embed a cost description provided by the cost model with the following interface:

```
$cost\_t$
    $enum\ rel$ { $better,\ worse,\ equal,\ unknown$ }
    COMPARE($otherCosts : cost\_t$) : $rel$

    COMPARETOTAL($otherCosts : cost\_t$) : $rel$
```

The method COMPARE compares the costs of one plan with the costs of another plan. Note that the cost model might be unable to decide which plan is cheaper, COMPARE can return *unknown* in this case. This does not happen when using a single number as cost description, but for a vector-based cost description a total ordering might be unavailable.

However, the potential lack of total order causes a problem when determining the optimal plan: For intermediate plans it is OK to keep different alternatives, but finally the optimizer has to decide which plan to execute. Therefore, the method COMPARETOTAL is used for the final plan comparison, which has to guarantee a total ordering. If it really cannot decide which plan is better it might choose an arbitrary one or might try to minimize some sub-goals like random I/O.

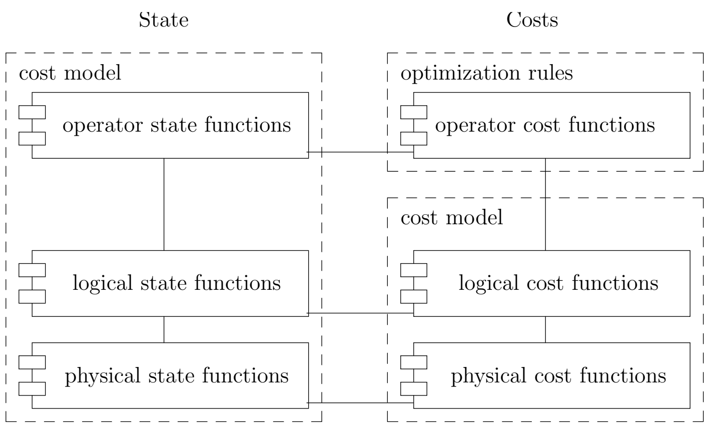

Figure 9.1.: Components of the cost model

9.3.3. Calculating Costs

As discussed above, costs are used to decide which plan is better than the other. However, these costs have to be calculated somehow, the simplest approach would be to just hide the calculations in the cost model, providing one method per operator:

tableScan(*segment*) : *cost_t*
nestedLoop(*leftCosts,rightCosts,rightSize*) : *cost_t*
hybridHashJoin(...) : *cost_t*

This approach has the advantage that every operation concerning costs is hidden in the cost model; the cost model can be used as a black box. However, is also has several disadvantages: First, the interface would be much more complex than sketched above. Even when only considering the primitive table scan, the cost model would need more information: How many pages should be used for prefetching? Is a full scan required or is it used as an existential quantifier? For more complex operators, the cost model requires much more information, making the calculation functions both hard to use and inefficient to call. Another disadvantage is that the cost model needs to know every operator supported by the plan generator, which makes adding new operators more cumbersome. Besides, this is a tight (semantic) coupling between the operators and the cost model, as the cost model has to understand the specific characteristics of an operator to calculate its costs.

This observation motivates a different cost model architecture, as shown in Figure 9.1. The cost model has to provide two sets of functions: One to keep track of a state like, e.g., cardinality, tuple size etc. (discussed in Section 9.3.4) and one to perform the actual cost calculations. The cost calculation can be divided in three layers: The lowest layer (*physical cost functions*) describes the hardware accesses. As this is very hardware specific we ignore it here, but a potential interface for a disk could be:

diskReadCosts(*diskId, sectorList*)

The function will most likely get a list of "expected" sectors (i.e. a list of distances between sectors) instead of real sectors, but still the function is very close to the hardware. The next layer (*logical cost functions*) uses these functions to describe logical operations, e.g. reading and writing a number of pages. Note that the function will need some parameters to identify the corresponding physical devices and to estimate the physical sector distribution (e.g. a segment id), but we simplify the inteface here.

```
readSequentialCost(pages) : cost_t
readRandomCost(pages) : cost_t
writeSequentialCost(pages) : cost_t
writeRandomCost(pages) : cost_t
```

Now these logical operations can be used to describe the operator logic in the topmost layer (*operator cost functions*). Note that this layer is not part of the cost model itself but part of the optimization rules. This way, the cost model is separated from the supported operators, while most of the cost model implementation is still hidden from the rest of the system. For example, a hash join might implement the following (simplified) cost function:

```
HashJoin::costs()
1  $leftPages \leftarrow$ calcPages($left$)
2  $rightPages \leftarrow$ calcPages($right$)
3  $partitions \leftarrow max(leftPages/memInPages, rightPages/memInPages)$
4  $leftCosts \leftarrow left$.costs() + writeRandomCosts($leftPages$)
5  $rightCosts \leftarrow right$.costs() + writeRandomCosts($rightPages$)
6  $partitionCosts \leftarrow$ readSequentialCosts($2 * partitions * memInPages$)
7  return $leftCosts + rightCosts + partitionCosts$
```

It first calculates the size of its input in pages and the number of partitions required to fit the input into main memory. During the join the data is first partitioned (resulting in random writes) and then the partitions are joined, resulting in sequential reads. The cost calculation itself is done by the cost model, the operator only calculates the physical characteristics.

For this approach, the exact interface provided by the cost model somewhat depends on the cost model itself, as different cost models might consider different physical properties. This is a disadvantage, but the dependencies are not too strong and the separation from the operators outweighs this limitation. A more detailed discussion of a complete interface is given in Section 9.4.

9.3.4. State

Whether the cost model performs the full cost calculation or just helps the operators to calculate the costs, some plan properties are required to perform the calculation. The most prominent is the cardinality. For some extremely simple cost models, this is actually the only property it needs, but usually it also needs some additional properties like the tuple size. Most of these properties are only relevant for the cost model itself and are just calculated and passed around to calculate the costs.

Therefore, these properties should be stored in an abstract data type whose actual implementation is only known to the cost model. However some (potentially derived) properties should be available through access methods, as they are useful to determine memory requirements etc. A simple interface is shown below:

```
state_t
    cardinality : double
    tupleSize : double
```

Note that the simple state shown above is not tied to a certain plan variant. For example, the join order influences the costs, but not the output cardinality. The same is true for other logical state information like data distribution. Hence, this state should not be stored in the plan itself but only once for each group of plan alternatives. However, the cost model might also consider physical information (like if and how the intermediate result is materialized) that is specific to a certain plan. In this case the state should be split into two parts: The logical part, that is common for all equivalent plans, and the physical part, which is stored inside the plan itself. We ignore this system-specific physical part in the rest of this work, as we abstract from a concrete database system.

The state is modified by the state functions shown in Figure 9.1. The lowest layer (*physical state functions*) modifies properties like the physical location, data distributions etc. However, as we abstract from the hardware in the simple state shown above, we ignore this layer here.

The next layer (*logical state functions*) updates the state according to logical operations. This is also very system dependent, as it needs information about data distributions etc., and is only used internally by the cost model. Therefore, we do not go into detail here. Typical operations are updating cardinalities, updating tuple information (after a projection or a concatenation) and estimating (dependent) selectivities.

The topmost layer (*operator state functions*) is the only layer visible to the rest of the system. It provides functions to initialize or update the state for each supported logical operator. Note that this layer is part of the cost model and not moved inside the optimization rules (like the operator cost functions). This is due to the fact that the number of logical operators is usually small (and complex logical operators can often be modelled by combining more simple operators), while the number of physical operators can be quite large. Two typical functions are shown below:

```
scanState(segmentId) : state_t
join(leftState,rightState,selectivity) : state_t
```

The method scanState initializes a state with the characteristics of a relation, and join combines two states into a new state describing the output of the join. Note that this interface reflects the logical algebra and not the physical algebra. That is, these state changes do not depend on the actual implementation of an operator but only on its semantic. Therefore, the number of these state functions is small.

Besides these functions for modelling logical operators, the cost model offers a function to enable cost calculations in DAGs: This more subtle function calculates the costs of the incoming operators. In the hash join example above this is simply done by using +, but this only produces the correct result for a certain class of cost models and for tree-structured operator graphs. This can be seen considering a simple example: When the right-hand side of a nested loop join contains a temp operator, it is not sufficient to multiply the right-hand costs with the cardinality of the left side. The calculation gets even more complicated when the left-hand side and the right-hand side share common operators, i.e., form a DAG. We will look into this in more detail in Section 9.4, but the cost model has to combine these costs in some way depending on the number of reads.

```
INPUTCOSTS(left,leftReads,right,rightReads) : cost_t
```

Note that depending on the implementation additional parameters are required to handle DAG-structured operator trees, but this is covered in the next section.

9.4. Implementation of a Cost Model

After discussing the required interface, we can now describe the actual implementation of the cost model. Since this work concentrates on supporting DAG-structured query graphs, the considered hardware characteristics are not very elaborated. However, this is not a fundamental limitation, more complex approximations could be integrated easily.

9.4.1. Data Structures

The cost description has to be embedded in each partial plan and has to allow fast comparisons with a small memory footprint. Therefore, we model the costs as a linear combination of I/O and CPU time [49] and just store the aggregated value. However, it is not enough to store one value per partial plan, many operators (in particular temp operators) have different characteristics when the data have to be read again. Multiple reads happen when using nested loop joins, which can be unavoidable when answering nested queries. The resulting cost description is shown below:

```
cost_t
    firstRead : double
    furtherReads : double

    enum rel { better, worse, equal, unknown }
    COMPARE(otherCosts) : rel
```

While this correctly models the behavior of different operators, it makes comparing plans difficult, as no total ordering among these costs exists. However, a partial ordering could result in a lot of plans that are not comparable, thus increasing the search space. In Section 9.6 we will look at the search space consequences of this model. Even if no total ordering exists, a partial ordering can be defined that allows comparing most of the plans, thereby reducing the increase in search space.

We now look at the problem of deciding if one plan (i.e. one cost description) is dominated by another one. The problem is that the cost description does not give a single value, but describes how the costs change depending on the number of reads, thus forms a function. The cost description can be considered as an affine function of the form $y = x * furtherReads + firstRead$ (y are the total costs for $x + 1$ reads). Now, given a set of plan alternatives, only those alternatives should be kept whose cost descriptions lay on the border of the hypograph of all cost descriptions. However, while calculating the hypograph is not very complex, it is too expensive to be done constantly during plan generation. Therefore, we just use a heuristic for comparison, which might state that a plan is not dominated although it is dominated by the other alternatives. Note that this still guarantees that the optimal solution can be found, it just increases the search space. The comparison method is shown below:

```
COMPARE($otherCosts$)
 1  if $firstRead < otherCosts.firstRead \wedge$
 2     $furtherReads \leq otherCost.furtherReads$
 3     then return $better$
 4  if $firstRead > otherCosts.firstRead \wedge$
 5     $futherReads \geq otherCosts.furtherReads$
 6     then return $worse$
 7  if $firstRead = otherCosts.firstRead$
 8     then if $furtherReads < otherCosts.furtherReads$
 9             then return $better$
10          if $furtherReads > otherCosts.furtherReads$
11             then return $worse$
12          return $equal$
13  return $unknown$
```

Note that *firstRead* is always equal to or greater than *furtherReads*. Therefore the comparison above orders actually most of the cost descriptions. Furthermore, there are only two common cases for *furtherReads*: Either *furtherReads* is equal to *firstRead* or *furtherReads* is less because of a materialization. In the latter case, *furtherReads* is the same for different plan alternatives, as the costs for reading a materialized result depend only on the cardinality and the tuple size. Therefore, the number of different cost descriptions is usually small.

Besides the costs itself, the cost model needs to maintain a certain state that is shared by the different plan alternatives. As discussed in Section 9.3.4, we only use a simplified state here, consisting of just the cardinality and the tuple size.

state_t
  cardinality : *double*
  tupleSize : *double*

  passes : *int*

The entry *passes* is used to keep track of multiple reads in combination with DAGs, it is neither used nor updated most of the time. Its usage is described in detail in Section 9.5.3.

9.4.2. Methods

In addition to these data structures, the cost model provides several methods to manipulate them. They were already discussed in Section 9.3, we only give a brief overview here. The cost functions visible to the rest of the system are:

readSequentialCost(*pages*) : *cost_t*
readRandomCost(*pages*) : *cost_t*
writeSequentialCost(*pages*) : *cost_t*
writeRandomCost(*pages*) : *cost_t*
cpuCosts(*cardinality,instructionList*) : *cost_t*

Further, some methods are provided to model the logical characteristics of the operators; they calculate the new state for each logical operator. The helper function spaceRequirements estimates the size of an intermediate result in bytes.

extentInfo(*segment*) : *state_t*
filter(*input,selectivity*) : *state_t*
join(*left,right,selectivity*) : *state_t*
map(*input,tupleIncrease*) : *state_t*
spaceRequirements(*state*) : *card_t*

Finally, binary operators need a method to determine the costs of reading their input. As this is not trivial when dealing with DAGs, it is described in detail in Section 9.5. Also note that the actual signature of the method is much more complex, but the basic interface is as follows (the *left*/*right* parameters are the input plans, the *leftReads*/*rightReads* parameters are the number of reads for each side):

inputCosts(*left, leftReads, right, rightReads*) : *cost_t*

9.4.3. Usage Example

Now all these methods are used by the different operators to model their runtime behavior. To illustrate this, we give a simplified code example of how the blockwise nested loop join calculates its state and costs:

```
BNLJoin::build($left, right$)
$result \leftarrow$ new $Plan()$

$passes \leftarrow \lceil \text{spaceRequirements}(right.state)/memSizeInBytes \rceil$
$examined \leftarrow left.state.cardinality * right.state.cardinality$

$result.state \leftarrow \text{join}(left, right, selectivity)$
$result.costs \leftarrow \text{inputCosts}(left.costs, 1, right.costs, passes)$
$result.costs \leftarrow result.costs + \text{cpuCosts}(examined, predicate)$
return $result$
```

As described in Chapter 5, the actual implementation is more complex because of the structure of the plan generator, but in principle, the different operators determine their behavior and then ask the cost model to calculate the costs and update the state. The cost model uses a weighted sum of I/O and CPU costs. However, this weighting is not visible in the code shown above, I/O and CPU costs are simply added. This is due to the fact that the weighting is done implicitly by adjusting the constants used to calculate the individual costs.

9.5. Algorithm

After choosing a concrete representation for the cost model, most cost model methods are straight-forward. While some care is needed to describe, e.g. access characteristics [82], the reasoning about costs itself is not very complex in classical plan generators. However, when supporting DAGs the calculation becomes much more complex. In this section, we first look at the simple tree case, then explain why DAGs are more difficult, and then present algorithms to calculate the costs for DAGs.

During plan generation, the costs are calculated incrementally. This means that the partial plans are annotated with the costs they cause, and when a rule creates a new partial plan it takes the costs caused by its input and adds the costs for the newly added operator. The costs for the operator itself are the same for trees and DAGs. However, the costs caused by the input can be different (due to sharing, as we will see below). Therefore, we only discuss how to calculate the costs caused by reading the input here. Note that this is only a problem for binary operators, as unary operators cannot construct DAGs (the input can be a DAG, but it can be treated as a scan with given costs). Therefore the cost calculation requires a function

inputCosts(*left, leftReads, right, rightReads*) : *cost_t*

that calculates the costs of reading *left leftReads* times and *right rightReads* times. This can than be used to calculate the costs for arbitrary binary operators.

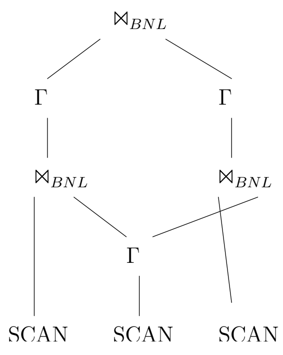

Figure 9.2.: DAG-structured query plan

9.5.1. Calculation for Trees

If it is clear that the operators actually form a tree (i.e. the two sub-graphs are disjoint; they themselves may form a DAG) the costs can be computed easily: Just multiply the costs with the number of reads and take into account that the first read might be more expensive.

```
INPUTCOSTSTREE($left, leftReads, right, rightReads$)
1  $leftCosts \leftarrow leftReads * left.costs.furtherReads$
2  $rightCosts \leftarrow rightReads * right.costs.furtherReads$
3  $leftDelta \leftarrow left.costs.firstRead - left.costs.furtherReads$
4  $rightDelta \leftarrow right.costs.firstRead - right.costs.furtherReads$
5  $result.furtherReads \leftarrow leftCosts + rightCosts$
6  $result.firstRead \leftarrow leftCosts + rightCosts + leftDelta + rightDelta$
7  return $result$
```

9.5.2. Problems when using DAGs

This simple approach does not work for DAGs. Consider the DAG-structured query plan in Figure 9.5.2. Here, the cost of the final blockwise nested loop join cannot be calculated in a straight-forward top-down calculation, in particular it cannot be determined by combining the cost of the two input operators. When the topmost join is treated like a normal join in an operator tree, the costs are overestimated, as the shared group-by operator is not taken into account. Since this operator serves two joins simultaneously, its costs should only be counted once. What complicates the calculation even more is that the shared operator appears twice on the right-hand side of a nested loop and, therefore, is read multiple times. The actual number of reads can only be determined by looking at all operators occurring in the plan. This makes calculating the costs complex and expensive and leads to the algorithm described below.

9.5.3. Calculation for DAGs

For DAGs, the calculation is much more complex than for trees. Shared subgraphs that are read by multiple operators have to be taken into account. For the cost calculation the actual number of passes (i.e. number of plan executions) has to be calculated, which can be lower than the number of reads in the case of sharing. At the same time, the partial plans involved must not be modified, as they could be used by other plans.

This requires some help from the operator rules involved, but can actually be performed efficiently without additional memory. The main idea is to use the *passes* entry embedded in the plan state to keep track of the number of reads (since this entry is reserved for this algorithm, it can be modified without disturbing other plans). The two input plans are traversed top-down. When visiting an operator for the first time, the costs are calculated as normal and the number of passes is stored. Further traversals can determine that the operator was already used and now only need to check if they require additional passes and calculate the costs accordingly. Some care is needed to accurately compute the number of passes, especially concerning (potential chains of) nested loops and materializing operators like temp.

The main function just delegates the work to the operator rules. Note that the code relies on the fact that the *passes* entry in each partial plan initially is equal to zero. This is always the case for new plans, and the algorithm described here, which modifies *passes*, resets it to zero afterwards. The algorithm guarantees that only nodes with *passes* $\neq 0$ can have children with *passes* $\neq 0$, enabling a linear runtime for resetPasses.

```
inputCostsDAG($left, leftReads, right, rightReads$)
1  $rc \leftarrow right.rule.$dagCosts($right, rightReads$)
2  $lc \leftarrow left.rule.$dagCosts($left, leftReads$)
3
4  resetPasses($left$)
5  resetPasses($right$)
6
7  $result.firstRead \leftarrow lc.firstRead + rc.firstRead$
8  $result.furtherReads \leftarrow lc.furtherreads + rc.furtherReads$
9  return $result$
```

```
resetPasses($plan$)
1  if $plan.passes \neq 0$
2     then $plan.passes \leftarrow 0$
3          for each $i$ input of $plan$
4             do resetPasses($i$)
```

Now the operator rules have to describe how the costs propagate through the operator tree. For the basic scan operations, this is trivial, it just needs to examine *passes* to detect shared subgraphs and to check if additional reads are required:

Scan::dagCosts($plan, reads$)

```
if $plan.passes = 0$
   then $result \leftarrow$ I/O costs for $reads$ passes
        $plan.passes \leftarrow reads$
        return $result$
if $reads > plan.passes$
   then $additional \leftarrow reads - plan.passess$
        $result.firstRead \leftarrow plan.cost.furtherReads * additional$
        $result.furtherReads \leftarrow plan.cost.furtherReads * additional$
        $plan.passes = reads$
        return $result$
return zero costs
```

Simple unary operators like a selection basically behave the same way, however, they have to re-calculate the costs of their subplan to propagate the number of passes.

```
SELECT::DAGCOSTS($plan, reads$)
if $plan.passes = 0$
   then $result \leftarrow input.rule.$DAGCOSTS($input, reads$)
        $result \leftarrow result +$ CPUCOSTS($reads, predicate$)
        $plans.passes \leftarrow reads$
        return $result$
if $reads > plan.passes$
   then $additional \leftarrow reads - plan.passes$
        $result \leftarrow input.rule.$DAGCOSTS($input, reads$)
        $result \leftarrow result +$ CPUCOSTS($additional, predicate$)
        $plan.pases \leftarrow reads$
        return $result$
return zero costs
```

Operators that materialize their results like a temp operator, have to calculate the costs of their subplans only once. However, they cannot just reuse the costs stored in the partial plan, as their input could become cheaper due to additional sharing introduced later on.

```
TEMP::DAGCOSTS($plan, reads$)
if $plan.passes = 0$
   then $result \leftarrow input.rule.$DAGCOSTS($input, reads$)
        $result \leftarrow result + reads *$ scan costs
        $plan.passes \leftarrow reads$
        return $result$
if $reads > plan.passes$
   then $additional \leftarrow reads - plan.passes$
        $result \leftarrow scancosts * additional$
        $plan.passes \leftarrow reads$
        return $result$
return zero costs
```

Binary operators basically behave like unary operators. However, they have

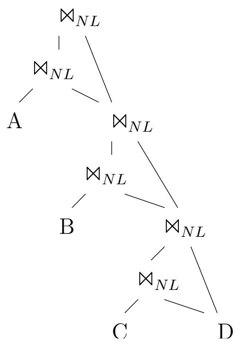

Figure 9.3.: A DAG requiring exponential runtime

to visit both input operators. As will be discussed in the next paragraph, the order of traversal is actually important.

```
NestedLoop::dagCosts($plan, reads$)
 1  if $plan.passes = 0$
 2     then $cross \leftarrow left.cardinality * right.cardinality$
 3           $result \leftarrow right.rule.$dagCosts$(right, reads * left.cardinality)$
 4           $result \leftarrow result + left.rule.$dagCosts$(left, reads)$
 5           $result \leftarrow result +$ cpuCosts$(reads * cross, predicate)$
 6           $plan.passes \leftarrow reads$
 7           return $result$
 8  if $reads > plan.passes$
 9     then $additional \leftarrow reads - plan.passes$
10           $cross \leftarrow left.cardinality * right.cardinality$
11           $result \leftarrow right.rule.$dagCosts$(right, reads * left.cardinality)$
12           $result \leftarrow result + left.rule.$dagCosts$(left, reads)$
13           $result \leftarrow result +$ cpuCosts$(additional * cross, predicate)$
14           $plan.passes \leftarrow reads$
15           return $result$
16  return zero costs
```

Note that the nested loop code shown can require an exponential runtime, as both input sources are visited (given a chain of nested loops where $left = right$, this could result in a runtime of 2^n). However, only the right input source is actually read multiple times. By visiting the right input first we make sure that the *passes* entry is set to a large value. Ideally, all further visits require a lower or equal number of passes, resulting in a linear time consumption.

While this is true most of the time, some extreme trees indeed trigger the exponential behavior. An example for this is shown in Figure 9.3. Depending on the selectivities and the cardinalities of the relations, the left-hand side might actually read the shared operators more often than the right-hand side: Assume that C consists of 100 tuples, D of 1 tuple and $C \bowtie D$ of 10 tuples; the joins are numbered 1 to 6 from top to bottom. During the cost calculation, $Join_5$ is asked to calculate its costs. As its left-hand side consists of 10 tuples, it asks D

for the costs of reading D 10 times. Afterwards, it asks $\bowtie_6$ for its costs, which detemines that D has to be read 100 times, which is larger than the previous value, requiring a revisit. The same situation can happen with B, visiting the right-hand side of $\bowtie_3$ twice and thus D four times (as the revisit increases the *passes* entries in the whole subgraph). The same happens for A, which leads to eight accesses to D, doubling with each additional nested loop pair.

In reality, this kind of DAGs do not occur, resulting in linear runtime. However, as they might occur and it is unsatisfactory to have an exponential step during cost calculation, we present two algorithms with linear bounds in the next sections. However, these algorithms are more involved and slower for most DAGs. Therefore, it might be preferable to try the exponential algorithm first. Since the linear case visits each plan node at most twice, it is safe to abort the algorithm after visiting more than $2n$ operators and switch to a linear algorithm. This guarantees both good best case and worst case performance.

9.5.4. Calculation in Linear Time

The problem with the algorithm described above is that plan nodes are visited multiple times and require a retraversal of their children if the number of passes increases. This potentially triggers a cascade of retraversals. Actually, this is not required. We now present an algorithm that computes the costs in quadratic time. This algorithm can then be transformed into an algorithm requiring linear time.

The operators that (potentially indirectly) produce the input of a certain operator o can be divided in two groups: The first group of operators is executed a constant number of times, independently of the actual number of reads of the output of o. The usual reason for this is that they are placed below memoizing operators like temp. The second group of operators is executed a number of times proportional to the number of reads of the output of o. It does not matter if these operators are read multiple times themselves, doubling the number of reads of o also doubles the number of reads of these operators. Note that these groups may overlap, as shared operators might be read both a fixed and a proportional number of times.

This observation leads to an algorithm that computes the costs in quadratic time and space. The idea is to compute the list of operators read in a partial plan together with the number of fixed and proportional reads for each operator. Thus, the list at the root of a DAG contains the number of reads for each operator in the DAG. The lists can be efficiently build bottom-up: For scans and unary operators this is trivial, binary operators can compute it by merging the lists of their two input operators. By using an arbitrary fixed total ordering of the operators, this merge can be done very efficiently, resulting in a total runtime of $O(n^2)$.

The algorithm is shown below, it stores the lists as *readOperators* in each partial plan. Note that the algorithm creates a temporary plan node as the root of the two input operators that behaves like a nested loop join with a given number of passes over each side. This is not strictly necessary, but avoids adding a special case to order the left and the right side. Besides, we use the function LOCALCOSTS to calculate the costs for n reads of one operator without the costs of its input.

```
INPUTCOSTSQUADRATIC($left, leftReads, right, rightReads$)
  $root \leftarrow$ create a new NLJOIN($left, leftReads, right, rightReads$)
  $list \leftarrow$ topological sort of $root$ and its subgraph
  for each $p$ in $list$ ( $backwards$)
    do $p.rule$.BUILDREADOPERATORS($p$)

  $result \leftarrow$ zero costs
  for each $t$ in $root.readOperators$
    do $result \leftarrow result + t.part$.LOCALCOSTS($t.fixed + t.proportional$)
  return $result$
```

We assume that the list of read operators consists of triplets [*part*, *fixed*, *proportional*], where *part* specifies the plan part, *fixed* the fixed number of reads of this plan part and *proportional* the number of reads that is proportional to the number of reads of the operator itself. For scans the list of read operators is simply empty:

```
TABLESCAN::BUILDREADOPERATORS($p$)
  $p.readOperators \leftarrow <>$
```

Simple unary operators like a selection just copy the list of their input and add the input itself:

```
SELECT::BUILDREADOPERATOR($p$)
  $p.readOperators \leftarrow$ MERGE($p.input.readOperators, < [p.input, 0, 1] >$)
```

Materializing operators like a temp operator behave the same, but they change the proportional number of reads into a fixed number of reads:

```
TEMP::BUILDREADOPERATOR($p$)
  $p.readOperators \leftarrow$ MERGE($p.input.readOperators, < [p.input, 0, 1] >$)
  for each $t$ in $p.readOperators$
    do $t.fixed \leftarrow t.fixed + t.proportional$
       $t.proportional \leftarrow 0$
```

Joins merge the lists of their input operators, adjusting the number of reads if required

```
NESTEDLOOP::BUILDREADOPERATORS($p$)
  $leftReads \leftarrow$ MERGE($p.left.readOperators, < [p.left, 0, 1] >$)
  $rightReads \leftarrow$ MERGE($p.right.readOperators, < [p.right, 0, 1] >$)
  for each $t$ in $rightReads$
    do $t.proportional \leftarrow t.proportional * p.left.cardinality$
  $p.readOperators \leftarrow$ MERGE($leftReads, rightReads$)
```

Finally, the merge step simply merges the lists by taking the maximum of *proportional* and *fixed*:

```
MERGE($l, r$)
  $result \leftarrow <>$
```

```
while $|l| > 0 \land |r| > 0$
  do $hl \leftarrow$ first entry of $l$
     $hr \leftarrow$ first entry of $r$
     if $hl.part < hr.part$
       then $result \leftarrow result \circ < hl >$
            $l \leftarrow l \setminus < hl >$
     if $hl.part > hr.part$
       then $result \leftarrow result \circ < hr >$
            $r \leftarrow r \setminus < hr >$
     if $hl.part = hr.part$
       then $f \leftarrow max(hl.fixed, hr.fixed)$
            $p \leftarrow max(hl.proportional, hr.proportional)$
            $result \leftarrow result \circ < [hl.part, f, p] >$
            $l \leftarrow l \setminus < hl >$
            $r \leftarrow r \setminus < hr >$
return $result \circ l \circ r$
```

So the algorithm scans the operator DAG in a bottom-up way to determine the operators transitively consumed by each operator. In the worst case, the plan is a list (e.g. n selections), resulting in quadratic runtime.

A nice property of this algorithm is that the *readOperators* lists do not depend on the parents of an operator and never change. Therefore, the lists can be maintained incrementally during plan generation, resulting in amortized linear runtime but quadratic space. This also eliminates the need for the topological sort, as the plans are constructed bottom-up anyway (even during a top-down search).

9.5.5. Calculation in Linear Time and Space

The algorithm described in Section 9.5.3 needs no additional memory besides linear space on the stack, but might require exponential runtime. The algorithm described in Section 9.5.4 guarantees linear time, but requires quadratic space. We now describe an algorithm that requires both linear time and space, with only somewhat larger constants than the algorithm requiring quadratic space.

The only reason why the first algorithm is exponential instead of linear is that a plan node might be visited again with a $reads > passes$. If we can always guarantee that this does not happen, each node is visited at most twice, resulting in a linear runtime.

This can be achieved be visiting the plan nodes in topological order: Each plan node passes the number of reads down to its children (iteratively, not recursively). Since the nodes are visited in topological order, the number of reads does not increase after the node is visited, resulting in linear time and only requiring linear space for the topological sort. The only disadvantage is that the topological sort has to be repeated for each cost calculation. Although this can be done in linear time, it results in larger constants than the incremental approach described in the previous section. A more detailed evaluation of the different algorithms is given in Section 9.6.1.

9.5.6. Full Algorithm

Regardless of the actual algorithm used, calculating the costs of a DAG is much more expensive than calculating the costs of a tree. Therefore, the cost model should avoid this expensive step whenever possible. This leads to a slightly different function than the one described in Section 9.3:

```
INPUTCOSTS($left, leftReads, right, rightReads, alternatives$)
if the subgraphs of $left$ and $right$ are disjoint
    then return INPUTCOSTTREE($left, leftReads, right, rightReads$)

$leftProp \leftarrow left.costs.furtherReads * (leftReads - 1)$
$rightProp \leftarrow right.costs.furtherReads * (rightReads - 1)$
$lowerBounds.firstRead \leftarrow max(left.costs.firstRead, right.cost.firstRead)$
$lowerBounds.furtherReads \leftarrow max(leftProp, rightProp)$
if $lowerBounds$ is dominated by one entry in $alternatives$
    then return $lowerBounds$

return INPUTCOSTDAG($left, leftReads, right, rightReads$)
```

At first the algorithm checks if the input plans overlap. This can be done easily by inspecting if the *sharing* properties of the input plans overlap, as they mark sharable operators present in the graphs. If they do not overlap, we have the cheap tree case. If they do, we can compute a lower bound for the actual costs by taking the maximum costs of each input. If this lower bound is already dominated by a known alternative, the cost calculation can be canceled. Only in the case that the plan seems interesting the expensive calculation is performed.

Note that the tree costs can always be used as an upper bound. This is useful e.g. for bounds based pruning.

9.6. Experimental Results

The cost model we presented has several implications on the runtime and the search space of the plan generator. Therefore, in this section we evaluate the impact of the different decisions and discuss some trade-offs. All experiments were executed on a 2 processor Intel Xeon 3 GHz Linux machine with 4GB main memory. The different algorithms were executed single threaded, so only one processor was used at a time.

9.6.1. Different Algorithms

In Section 9.5 we presented three different algorithms to calculate the costs for a DAG with different time and space characteristics. We analyzed their runtime behavior by looking at different cases.

First, we constructed a simple left-deep tree containing nested loop joins and used the DAG algorithms to calculate the total costs. While in this case a DAG algorithm is not actually required since the input is a tree, this problem gives a reasonable lower bound for the runtime overhead for few shared operators.

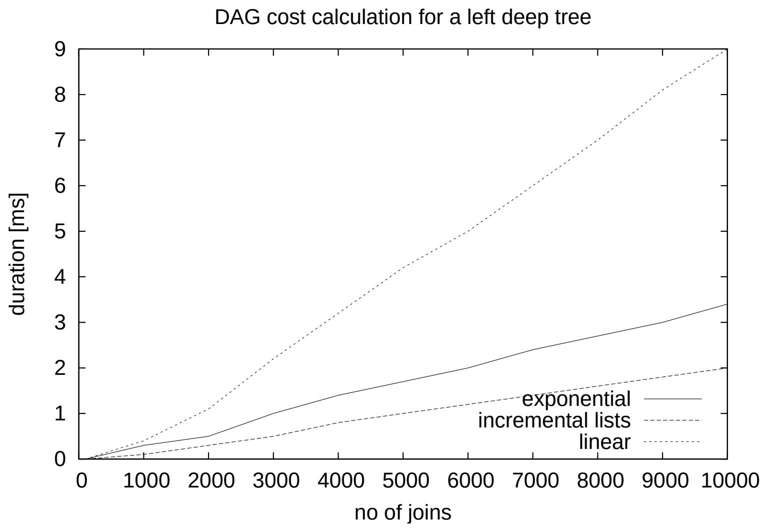

Figure 9.4.: DAG calculation costs for a left-deep tree

The results are shown in Figure 9.4. All algorithms need a linear time for this problem, the algorithm using lists being the fastest and the algorithm with guaranteed linear time and space the slowest. This is due to the fact that the algorithm needs to calculate a topological sort, which needs about the same time as the actual cost calculation. Note that the absolute time for the cost calculation is quite small, especially since most queries will consist of perhaps 50 but not 1000 operators. However, measuring such small problems would show only noise, and since the algorithms are linear (at least in this example) the results for large problems show what happens when the algorithms are executed multiple times.

After a tree without any shared operators, we examine a stack of nested loop join operators, where both the left and the right side of a join reads from the next join below. This is an example with a massive amount of operator sharing and would trigger an exponential behavior with a naive cost calculation. The results are shown in Figure 9.5. The algorithms perform similar to the left-deep tree, so operator sharing does not have a large influence on cost calculation. Note that the absolute time is lower than for the left-deep tree, as the join stack includes less operators (the left-deep tree has tablescans on the right-hand side of each join).

Finally, we looked at the example DAG shown in Figure 9.3, which triggers the exponential behavior in the first algorithm. The results are shown in Figure 9.6. Here the last two algorithms again need linear time and behave very similar, however, the first algorithm indeed starts to require exponential time. This trend only stops after 3000 operators due to hardware limitations: The floating point numbers used to represent *passes* can no longer detect the need to retraverse. In this example, the runtime of the first algorithm is clearly unacceptable, although the absolute time for the more realistic case of 100 operators is not too bad.

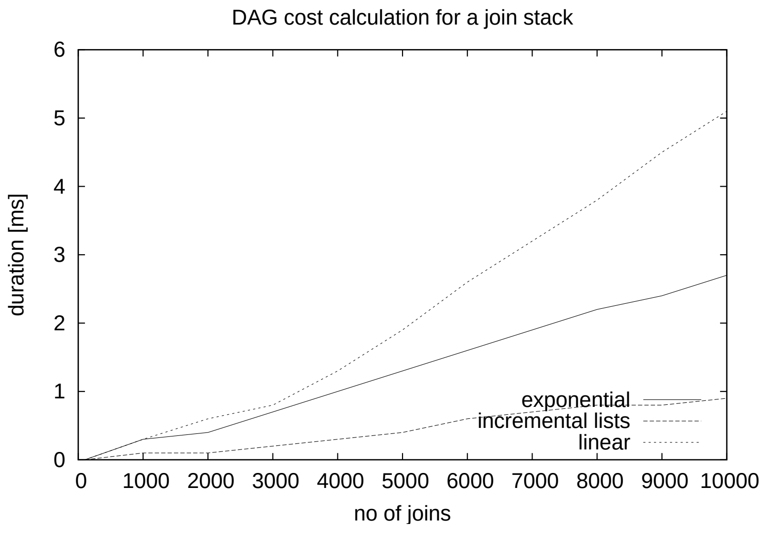

Figure 9.5.: DAG calculation costs for a stack of joins

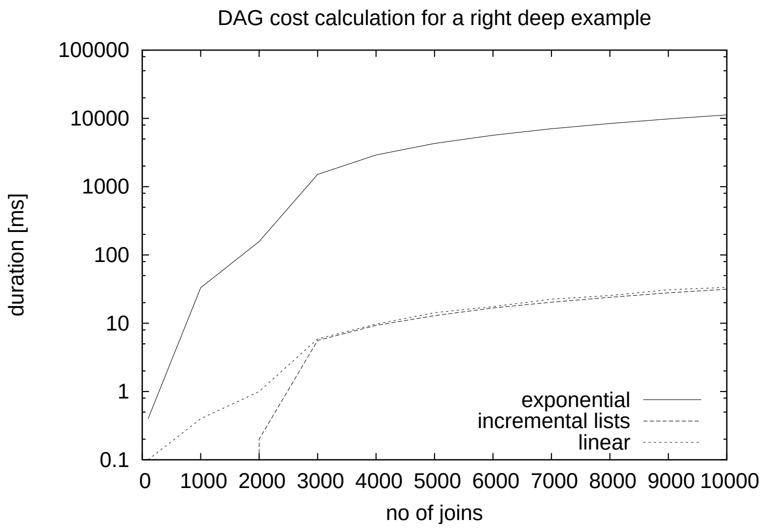

Figure 9.6.: DAG calculation costs for a right-deep tree, as shown in Figure 9.3

For all examples, the incremental list algorithm was clearly the fastest. However, it requires a quadratic amount of space, which is quite large: For 1000 operators, it needs 19 MB and for 10000 operators 1.9 GB! While the more realistic example of 50 operators only requires 48 KB, this is still 1 KB per partial plan, which is too much for today's main memory. While the memory could be released after the cost calculation, the repeated calculation would require quadratic time and performs much slower than the other algorithms. So while the incremental list algorithm is the fastest, its space requirements will probably prevent its usage in the near future.

The other algorithms are somewhat slower but only need linear space (one counter per plan node and enough stack space to visit all plan nodes). Most of the time the exponential algorithm is faster than the linear algorithm, but of course this changes dramatically when the exponential case is triggered. On the other hand, for 100 operators even the exponential case is only about a factor of 10 slower than the linear algorithm, and twice as fast in the normal case. So if the exponential case is sufficiently rare it might still be worthwhile to use the exponential algorithm. The linear algorithm is a safe choice, it needs as little space as the exponential algorithm and guarantees linear runtime with a slowdown of about a factor of 2 compared to the linear case of the exponential algorithm. This factor will even decrease when the cost functions themselves become more expensive (e.g. when modeling the hardware more detailed), so the linear algorithm might actually be the best choice.

9.7. Conclusion

We described a cost model suitable to handle DAGs and presented three different algorithms to calculate the costs of executing a DAG-structured plan. While the first two algorithms had some problems concerning runtime and space requirements respectively, the third algorithm guarantees execution in both linear time and space. Future work should examine if it is worthwhile to use the usually faster exponential algorithm, if the exponential case is sufficiently rare.

10. Execution of DAG-Structured query graphs

10.1. Introduction

While it is clear that DAG-structured query plans are superior to purely tree-structured query plans from a theoretical point of view, the practical situation is not so obvious. One surprisingly difficult problem when dealing with DAG-structured query plans is the actual execution.

Consider the query plans in Figure 10.1. In both plans the subqueries A, B and C are joined in a way that joins B twice, once with A and once with C. In the tree case this is done by duplicating the subquery B, which means duplicate work for B. The DAG plan can avoid this duplication, as B can be simply shared by the join with A and the join with C. The tree plan can be executed by using the standard iterator model [50]: Recursively, each operator pulls the required tuples from its children until the topmost join has computed the query result. In the DAG case this is not possible, as the tuples from B are required by two different operators: if one of them fetches a tuple it might be missed by the other one.

When such a situation occurs (e.g. when using views), it is usually solved by spooling B to disk, which eliminates the problem, as now the two joins can read the data independently of each other. But spooling to disk is inefficient, and also fails to handle another class of DAG-structured query plans: Figure 10.2 shows a bypass plan [73], where tuples are treated differently depending on whether they satisfy a condition or not. While this could also be handled by spooling the different streams to disk, the cost would be too high, as bypass splits can also be done for extremely cheap operations like selections.

We will discuss the different existing approaches in more detail in Section 10.3, but neither of them is really satisfactory. A general execution strategy for DAG-structured query plans should satisfy the following goals:

1. it must handle arbitrary DAGs

2. it should not introduce new pipeline breakers

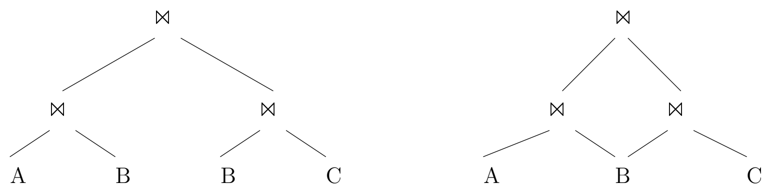

Figure 10.1.: A query plan as tree and as DAG

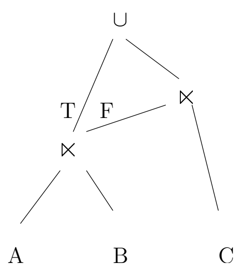

Figure 10.2.: A bypass plan for disjunctive conditions

3. it should have minimal overhead, especially for trees

The last goal is very important, as most of the generated plans will probably form trees and, therefore, a slowdown for this common case is unacceptable.

In the rest of this chapter we first discuss the different alternatives to execute DAG-structured plans and then present a new approach that is better suited for general DAGs. Experimental results show that the overhead for supporting DAGs is acceptable. This means that even for ordinary tree plans an implementation which supports also DAGs is barely slower than an implementation that only supports trees. This makes DAG support more attractive.

10.2. Related Work

After the standard iterator model for operators has been established [50], few other papers discuss an evaluation model for query plans. While some papers create DAG-structured query plans [5, 29, 73], none of them discuss how they should actually be executed.

The execution of a limited form of a DAGs is discussed in [21]. It describes the parallel execution of operators as performance improvement: A tuple stream is split in multiple streams, each stream is handled by an operator executed in parallel to the rest and the results are joined afterwards. This is especially intuitive for sorting, but can also be done for other operators. But the DAG structure is very confined and the data is always partitioned, not passed to multiple operators.

One interesting approach is the Telegraph project [4, 12]. There, tuples do not pass through a classical operator tree, but are passed individually between operators, potentially even visiting operators multiple times. The idea is that the execution should adapt to changing data characteristics. By adapting the data flow on the fly, the runtime system can change wrong decisions made at compilation time. This is important when little is known about the actual data, as in stream processing. While the papers do not cover DAG-structured query plans, it might be possible to use this approach also for DAGs, as it supports arbitrary rerouting of tuples. However, this would be much more general (and probably with a larger overhead) than the approach described here.

An architecture that explicitly handles DAG-structured query plans is described in [42]. It uses a push approach, where the data is passed from producers to their consumers. Depending on the operator structure, the operator

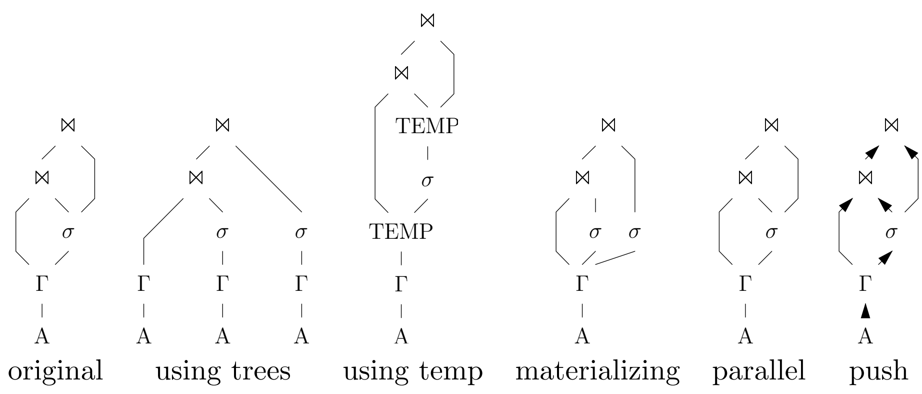

Figure 10.3.: Different execution strategies

groups either run in their own threads, form a simple pipeline or use buffering to handle multiple consumers. Currently, not many details about this have been published.

10.3. Execution Strategies

While a direct execution of a DAG-structured query plan is usually not possible in a classical database management system, there still exist strategies to execute these plans, even in a system designed for trees. In the following, we present some of these strategies, beginning with the least invasive and ending with a very invasive but also very efficient execution strategy. As an illustrating example, we use the execution plan shown on the left-hand side of Figure 10.3.

10.3.1. Using Trees

The simplest way to execute a DAG-structured query plan is to first convert it into a tree. This is done bottom-up by creating a copy of every shared tree until all trees have unique parents. For our example, the result is shown in the second column of Figure 10.3.

This strategy is not really an option. While it can handle arbitrary plans, it eliminates all advantages of the original DAG plan. Especially for bypass plans [73], the resulting tree is probably worse than a plan without bypass functionality.

10.3.2. Using Temp

The reason why a normal database system cannot execute DAGs directly is that the same data has to be read multiple times by different consumers, which does not work if the data is only passed between operators. However, what these systems usually support is reading the same data for different consumers if the data is stored on disk (e.g. in a relation). This can be used to execute DAGs: Every operator that is read by multiple operators must be a scan (that is a relation, an index or a `temp` operator that spools its input to disk). The transformed plan for our example is shown in the third column of Figure 10.3: Both the output of the group-by and the output of the selection are spooled to disk, as they are read by multiple operators.

While this strategy is often better than converting the DAG into a tree, the overhead for the `temp` operator is considerable. For the group-by in our example it might be still be advantageous (as a group-by is expensive and also reduces the cardinality), but for the selection it is probably not.

Note that a variation of this strategy is buffering: Instead of spooling the result to disk, we keep it first in a memory buffer and remove all data that has been seen by all consumers. If the access pattern of the consumers is "nice" (i.e. they read the data more or less synchronously) this can avoid the expensive spooling to disk. However, if they differ too much this scheme falls back to the normal `temp` approach.

10.3.3. Share Only Materializing Operators

The problem with the previous strategy was the overhead for the additional `temp` operators, as the data has to be read and written at least one more time. However, many operators (especially the expensive ones) spool the result or at least intermediate results to disk anyway: The sort or group-by operator, for example, have to go to disk if the data is too large, otherwise they keep the whole data in memory. In both cases, it is possible to compute the result multiple times without reading the input multiple times. The operators that do not spool to disk are usually much cheaper (but a counter example is shown below) and can, therefore, be executed multiple times without too much additional work.

So this strategy is a combination of the previous two strategies: Shared operators that materialize their result can be shared directly and the other operators must be duplicated. The result for our example can be seen in the fourth column of Figure 10.3. This approach avoids the overhead of `temp` operators and still allows for sharing partial results. While some minor modifications might be required to allow for multiple readers of operators, this strategy is probably the best compromise possible without major changes in an existing database management system. The disadvantage is that not all operators can be shared. While most of the operators that do not materialize are cheap, some can be very expensive (e.g. a `djoin`). Then it might make sense to add a `temp` operator, of course with the same overhead as the previous strategy.

10.3.4. Parallel Execution

A completely different evaluation strategy that also supports DAGs is the parallel execution of operators: If every operator is a separate process (possibly even on a different computer), multiple readers are usually not a problem: As they have to synchronize their work anyway, either by a simple rendezvous protocol or by some more elaborate means, synchronizing with more than one reader is not very different. This kind of execution has been done in the past by distributed or parallel systems [21].

The disadvantage of this strategy is that overhead for passing data between operators is quite high. This might not be a problem for distributed or parallel systems, but for a single processor system executing the plans in this way is quite wasteful. And even for a distributed system it makes sense to execute the local part of a plan with reduced overhead.

10.3.5. Pushing

The main problem with executing DAGs is that the same data is consumed by multiple operators. Using the standard iterator model [50], this means that multiple consumers want to iterate over the same data independently, which usually cannot be done without buffering or spooling. This problem can be avoided by reversing the control flow: Instead of the operators iterating over their input, the input "pushes" the data up the DAG, i.e., when an operator has produced some data, it hands it to all its consumers at the same time. This is shown on the right-hand side of Figure 10.3.

When using a push model, an arbitrary number of consumers can be served at the same time without any buffering or spooling. In a way, this is similar to the rendezvous protocol used for parallel execution, as each operator notifies its consumers of available data. A problem of this approach is that the operators no longer have full control of the speed in which they get their data (e.g. in our example, the left-most join gets data from both input sides at the same time, which is a problem for nested loop joins), but as we will see in the rest of this chapter, this can be handled. The great advantage is that DAGs with arbitrary operators can be handled with minimal overhead and without copying data.

10.4. Algebraic Operators

Algebraic operators are used during query execution to compute and combine partial results until the whole query has been answered. Both this execution model and the interface offered by the operators have remained essentially unchanged since the beginning of relational database systems [50]. However, they do not support DAG-structured query plans very well. Of the execution strategies discussed in Section 10.3, nearly all of them require at least minor changes to the standard model. Especially the push strategy, which offers the most generic and efficient execution, is not supported very well. In this section, we first look at the standard model of algebraic operators, then discuss how it has to be changed to support a push execution and then consider the concrete interface that should be offered. The required changes should be as minimally invasive as possible, as a large number of operators has already been implemented and existing systems will probably support DAGs only if it can be done without too much work. Therefore, we also briefly discuss changes required for existing operators.

10.4.1. Existing Interface

The standard (pull) operator interface is very simple:

```
Operator
    OPEN() : void
    NEXT() : boolean
    CLOSE() : void
```

The method OPEN initializes the operator, allocates required resources and prepares the operator to produce the first tuple. The method NEXT produces the next tuple if any, and returns *false* when all tuples have been produced. The method CLOSE finally releases all resources required for the operator. So the operator is used as an iterator over its output, each NEXT call produces the next entry. Note that this interface hides the actual passing of tuples, it is assumed that they are stored somewhere else and are accessible by the other operators. So the operator interface is mainly concerned with the control flow, not the data flow.

This interface is only useful for trees. When the NEXT method is called, the operator has no way to determine who wants to get the next tuple. Thus, it is unable to serve multiple consumers, as it cannot decide when to produce the same tuple again and when a new one. While this could be solved by passing a parameter, it could require buffering the whole output, which is highly undesirable. One solution for this is changing from a pull model, where every operator requests its input, to a push model, where the operators report their output to the consumers.

10.4.2. A Push Interface

When the data is pushed bottom-up instead of pulled, the operators can no longer request data from their input. Instead, they have to wait for events, either that new data has arrived (which is then processed, potentially creating new events) or that all data has been produced and the computation can be finished. Thus, the operators provide two callback methods used for notification:

```
Operator
    DATAEVENT(source : Operator) : void
    ENDOFDATAEVENT(source : Operator) : void
```

When an operator has produced a tuple, it calls the DATAEVENT method of all its consumers; the parameter *source* specifies the producing operator. Similarly, the ENDOFDATAEVENT method is used to notify the end of data.

While this basic interface is enough to pass data between operators, it is not enough to execute a whole query. Consider, for example, a query with two scans and one join. To execute this in a pull model, the NEXT method of the root (the join) is called until all data has been produced. However, in a push model some operator other than the root has to start producing data, and for the join both scans have to produce data, preferable in a sequence beneficial for the join. We will look at the details of this in Section 10.5. For now we just assume that an operator can *activate* one of its input operators, which causes it to produce data in the near future (not necessary immediately).

To illustrate this interface, we consider two operators: A selection and a hash join. Note that these descriptions are very high level, a more detailed discussion can be found in Section 10.5.

```
SELECTION::DATAEVENT(source)
1  if the data satisfy the predicate
2     then call DATAEVENT(this) in all consumers
```

```
SELECTION::ENDOFDATAEVENT(source)
1  call ENDOFDATAEVENT(this) in all consumers
```

When the selection receives a data event, it checks the selection predicate and passes the event to its consumers if the predicate is satisfied. The end of data events are passed unconditionally. The hash join is more complex as it has to check from which side the data came.

```
HASHJOIN::DATAEVENT(source)
1  if $source = left$
2     then store the data in the left hash table
3  if $source = right$
4     then store the data in the right hash table
```

When the hash join gets a data event, it just takes the data and stores it in a hash table. Note that the event is never passed to its consumers. This only happens after all data has arrived:

```
HASHJOIN::ENDOFDATAEVENT(source)
1  if $source = left$
2     then activate the right input
3  if $source = right$
4     then join the hash tables
5          for all matches
6             do call DATAEVENT(this) in all consumers
7          call ENDOFDATAEVENT(this) in all consumers
```

Here we assume that the left-hand side produces data first (this assumption can be eliminated easily). After the left-hand side has produced all data, the join activates the right-hand side, as it also has to produce data. After the right-hand side is finished, too, the join operator combines the two hash tables and produces a data event for all matches. Finally, it notifies its consumers that all data has been produced.

10.4.3. Reusing Existing Operators

While the push interface looks very different from the pull interface, converting a pull operator into a push operator is usually not very complex. Two aspects have to be changed: Passing data up the tree and getting data from the input.

In the pull model data is passed by returning `true`/`false`, in the push model the DATAEVENT/ENDOFDATAEVENT methods are called instead. In the pull model the operators call NEXT to get data, in the push model the operators

ACTIVATE the input operator and leave the current method. These changes are very minor, the main difficulty is the fact that in the push model the control flow changes, the method ends after activating an input operator and only continues when data arrives. Still, most code can be reused directly.

10.5. Implementation Details

The main problem of the push model is the control flow. First it has to determine which operator should produce data first. Then this operator pushes data up to its consumers, until, e.g., a join decides that another operator should produce data. Now the control flow should change, but the current operator might not be finished yet, so the control flow might change back to the current operator later on.

When implementing this naively, this control flow change cannot be done with common imperative languages like C or C++. The operators receive their data from their callers, so in order to receive data from other operators an operator would have to change its callers, which basically means changing the past. In the rest of this section we sketch two implementation alternatives which solve this problem. The first uses coroutines to manage this change of control flow. However, this is not well supported by most programming languages and is quite expensive. Therefore, the second alternative changes the control flow by explicit scheduling.

10.5.1. Coroutines

The most intuitive way to implement the push model is to use coroutines (or threads/processes). Each operator runs in its own context, waits for input and tries to produce output. An operator can produce output if it has the required input and its consumers accept the output, so they implement a simple rendezvous protocol. Here an operator literally activates another operator: it switches to the context of the other operator.

This model facilitates the implementation of operators. The code shown below does not use the event interface from Section 10.1, as it is more natural to write coroutines this way, but it could be split into multiple methods so that the event interface remains.

```
HashJoin::routine()
 1  while true
 2    do activate the left operator
 3       if it reported end of data
 4         then break
 5       store the data in the left hash table
 6  while true
 7    do activate the right operator
 8       if it reported end of data
 9         then break
10       store the data in the right hash table
11
```

```
join the hash tables
for all matches
  do activate all consumers, report data

while true
  do activate all consumers, report end of data
```

When using coroutines, the operators can be written straightforwardly. All operations are written as in a pull model and the result is just passed up to the consumers. This also eliminates the problem of which operator should push first: Just activate the root operator, it activates its input as needed.

The code shown above is very simplified, there is some infrastructure required to support multiple reads or operators that do not expect input from a source yet (imagine that the hash join gets input without ever been activated first). But these are just details, the infrastructure is shared by all operators and the operators themselves are quite simple. The main disadvantage of the coroutine model is that it is too slow. We will look at some timing results in Section 10.6, but the context switches involved in switching from one operator to another are very expensive. Therefore, we developed a much faster, but also more complex alternative.

10.5.2. Push by Pull

As switching to another coroutine is very expensive, we constructed a solution that can be implemented with normal function calls. The implementation sketched here uses an explicit scheduling mechanism that solves the control flow problems and can be implemented using standard programming constructs. Although this provides a push model, the scheduling itself is done in a very pull-like way (as we will see below). Therefore, we called this method "push by pull".

The operators receive data as events and they produce new data, creating events for other operators. This results in a very complex control flow. It can be formulated easily by using coroutines, but even without coroutines it can be done by explicitly scheduling the events. For example, it would be possible to organize all events in a priority queue using some criteria and during query processing always remove the most important event, activate the corresponding operator and enqueue the newly produced events. Such a scheme allows for arbitrarily complex data flow, and is easy to implement in standard programming languages. However, the overhead is very high compared with pull model: the data associated with the data events has to be materialized if operators place more than one event at the same time into the queue and also the queue management itself consumes CPU.

To avoid this overhead, we restrict the scheduling in two ways. First, operators only produce new data events after their existing data events have been consumed (this avoids materializing). Second, the control flow changes only if it has to because of missing data (this reduces the scheduling costs). In practice, this means that operators trigger their consumers (with a direct method call) as long as possible. For example, consider a blockwise nested loop join between two tablescans. The left scan starts producing data and pushes it into the join. The join consumes this data (by storing it in a buffer), but otherwise does nothing and allows the scan to continue producing data as long as the buffer is not full. If the buffer is full, the right scan must produce data, which means that the left scan must stop. Thus, when handling an event, an operator reports if its producer should stop or continue producing data. If the producer stops, a scheduling component selects the next operator that should produce data.

This model requires several minor modifications to the push interface described in Section 10.4.2. First, there is a separate scheduling component: The scheduler selects an operator that should produce data and this operator pushes its data up to its consumers. Second, the event methods (DATAEVENT / ENDOFDATAEVENT) can return a boolean value. If one operator requires a reschedule because it needs input from some other operator, it simply returns `false` as the result of an event. If the event was created from within another event method, this method also returns `false` etc., until the control flow reaches the scheduler which triggered the first event. The scheduler now selects the next operator and data is produced until the next reschedule is required etc. This way, the scheduler is only activated as needed, reducing the overhead to a minimum. The extended interface is shown below:

```
Operator
    activateNext : Operator

    ACTIVATE(source,newSource : Operator) : boolean
    REPORTDATA() : boolean
    REPORTENDOFDATA() : boolean

    DATAEVENT(source : Operator) : boolean
    ENDOFDATAEVENT(source : Operator) : boolean
    STARTPUSH() : void
```

The attribute *activateNext* is a hint for the scheduler: if it is set it points to the operator that should be activated instead of this operator (which usually means that the other operator has to produce data for the current operator first). It is set by the ACTIVATE method, that activates a requested operator:

```
OPERATOR::ACTIVATE($source$, $newSource$)
1  $activateNext \leftarrow newSource$
2  return $source = newSource$
```

It sets *activateNext* and checks if the requested operator is the same as the source of the current event. If not, it returns `false`, which causes all callers to drop back to the scheduler, which can now activate the proper operator.

When an operator creates new data or reaches the end of data, it has to notify its consumers. This is done by the small helper functions REPORTDATA and REPORTENDOFDATA that trigger the corresponding events in the consumers of the current operator. If any of the event functions return `false` (i.e. request a reschedule), the report functions also return false, which triggers a fallback to the scheduler. The pseudo code is shown below:

```
Operator::reportData()
1  result ← true
2  for each c in consumers
3    do if c.dataEvent(this) = false
4         then result ← false
5  return result
```

The reportEndOfData function is nearly identical, it calls endOfDataEvent instead and clears *activateNext*, as no input is required after all data has been produced.

The dataEvent and endOfDataEvent functions were already discussed in Section 10.4.2, the additional return value is used to request a reschedule. The new startPush method is called by the scheduler if it determines that the operator should start producing data.

To illustrate the mechanism, consider the following simple selection operator:

```
Selection::dataEvent(source)
1  if the data satisfies the predicate
2    then if reportData() = false
3           then return false
4  return activate(source, input)
```

```
Selection::endOfDataEvent(source)
1  while true
2    do if reportData() = false
3         then return false
```

```
Selection::startPush()
1  activate(nil, input)
```

For a selection the startPush method makes no sense, as it always requires data. It simply activates its input. If it gets data, it checks the predicate, and if this is satisfied, it pushes the data up using reportData. If one of its consumers requests a reschedule, it drops back to the scheduling component, otherwise it uses activate to request more data. If it gets an end of input event, it simply pushes this fact upwards until some operator requests a reschedule.

Binary operators are more complex. We consider here a simple hash join:

```
HashJoin::dataEvent(source)
1  if source = left
2    then store the data in the left hash table
3         return activate(source, left)
4  if source = right
5    then store the data in the right hash table
6         return activate(source, right)
```

```
HashJoin::endOfDataEvent($source$)
if $source = left$
  then return activate($source, right$)
if $source = right$
  then $activateNext \leftarrow$ nil
       join the hash tables
       for all matches
         do if reportData() = false
              then return false
       while true
         do if reportEndOfData() = false
              then return false
```

```
HashJoin::startPush()
if already joining
  then continue joining the hash tables
       for all matches
         do if reportData() = false
              then return

activate(nil, $left$)
```

The dataEvent and endOfDataEvent methods are nearly identical to the ones from Section 10.4.2, the only interesting detail is that *activateNext* is reset before joining the hash tables, as the operator does not need input anymore. The startPush method is called when the scheduler has determined that the operator should produce data. Then, there are two cases: Either the operator is already joining the entries of the hash table (in which case it continues to do so), or it requires more input, in which case it activates its left input operator (this is somewhat arbitrary, it could start with the right hand side as well).

The scheduling component required is very simple, it just tries to activate the root of the query plan until the whole result has been produced:

```
Scheduler::run()
while $root$ did not get an end of data event
  do $iter \leftarrow root$
     while $iter.activateNext \neq$ nil
       do $iter \leftarrow iter.activateNext$
     $iter$.startPush()
```

This causes the pull-like scheduling: When an operator needs input from another operator, it sets *activateNext* and falls back to the scheduler. This causes an execution order similar to the pull model.

10.5.3. Scheduling

While the scheduling algorithm shown above works, it is very simple. It does not try to satisfy any goal besides correctness, especially it ignores any resource consumption. However, when a query plan includes pipeline breakers, there are usually different scheduling alternatives. For example, the hash join shown above could fall back to the scheduler after getting all data instead of directly producing matches. Then, all input operators for the join could release their resources, the scheduler could activate some other part of the query plan, and later on the join would be activated again to produce the matches for another operator. This might result in a much more conservative resource usage, especially for main memory.

In this work we concentrated on query optimization and, therefore, ignored more advanced scheduling techniques, but if the runtime system uses explicit scheduling anyway, it might be worthwhile to make use of it.

10.5.4. Possible Restrictions

While the push model supports arbitrary query plans – in principle –, there is a restriction when using the standard operators naively: A binary operator must not read the same operator directly (i.e. without pipeline breaker in between) twice. Consider a plan that consists just of a tablescan and a nested loop join that reads the table both on its left and its right-hand side. When the scan pushes its output up, the join gets data on both its left and its right hand side at the same time and not n tuples on the right-hand side for each tuple on the left-hand side. Blocking the left-hand side does not help, as the left-hand side is the same as the right-hand side. Note, however, that it is possible to execute the nested loop join in this plan: It must just ignore the incoming tuples on the left-hand side, the right-hand side is read multiple times anyway so the left-hand side is also regenerated. With a minimal amount of buffering (the next tuple on the left-hand side), this allows executing the nested loop join in a push way without any additional work. However, it requires changing the join implementation.

The easiest way to avoid these problems is to make sure that there is a pipeline breaker between the shared source and the join. The pipeline breaker can accept input without producing output. Therefore, it allows to decouple the join from the source, the join can read the data in the order it wants. As many operators (sort, group-by, grace hash join etc.) are pipeline breakers, this is not a very severe limitation.

However, more operators than just the nested loop join can be adjusted to accept reading the same source twice. A similar trick can be done for a blockwise nested loop join (it gets spurious tuples when filling the memory buffer, but it knows that it has to join all entries in the buffer anyway) and, in fact, all standard binary operators can be adapted to handle this problem: a sort merge join, for example, either wants to perform the join on the same attribute (which, in fact, allows for a more efficient implementation) or it has to add at least one sort operator anyway, which is a pipeline breaker. The same is true for hash-based join operators that require a partitioning enforcer. While theoretically some operators might exist that cannot be changed this way, the push model does not impose restrictions for the currently known operators.

10.6. Experimental Results

To evaluate the different execution strategies, we implemented the basic runtime operators in three different versions: First as a classical pull model, then as a push model using coroutines, and finally as push-by-pull model as described in Section 10.5. Note that the same basic algorithms were used for all three versions. Of course the pull version cannot execute DAG-structured query plans. However, it is the most common model used today and serves as a base line to estimate the overhead for DAG support. The coroutine version represents the parallel execution strategy; instead of coroutines, it would be possible to use threads or processes. But as coroutines have a lower overhead than threads or processes, this implementation was chosen to consider a reasonable fast implementation. The push-by-pull version is what we finally propose as an efficient execution model to support DAGs.

The experimental results shown below were all performed on a 2.4 GHz Pentium IV, using the gcc 3.3.1.

10.6.1. Data Passing Overhead

To evaluate if the overhead for DAGs is not too high, we first measured the overhead required to pass data between different operators. Therefore, we constructed query plans that consist of a single main memory table scan of 100 tuples and a sequence of n selections, which all have *true* as a selection predicate. When executing this plan, the data is fetched from the table and then passed with minimal additional work through the sequence of selection operators. So we basically measure the cost of passing data between operators.

The results are shown in Figure 10.4. The overhead when using coroutines is very high, it is at least a factor of 20 compared to the basic pull implementation. Of course this does not mean that real queries will also be slower by a factor of 20, as this test only measures data passing, but still the coupling between the operators is expensive. Interestingly, the push-by-pull method seems to be actually faster than the basic pull method when the number of selections is small. However, this is somewhat misleading: This is mainly because only 100 tuples were read and the initialization is faster, as no *open*/*close* calls are required. The absolute time difference is very small. Still, even for a large number of operators the push-by-pull approach performs very well, with only a very minor overhead compared to the pull approach.

As few queries contain even 100 selections, we also considered a query plan with only 10 selections executed after a file scan with an increasing number of tuples. The results are shown in Figure 10.5. Again, coroutines are very expensive, but the difference between pull and push-by-pull is very small, as the scan itself is expensive compared to a simple selection.

10.6.2. Control Flow Overhead

The previous experiment has measured the overhead for passing data between operators, but it has a very primitive control flow, the scheduler in the push-

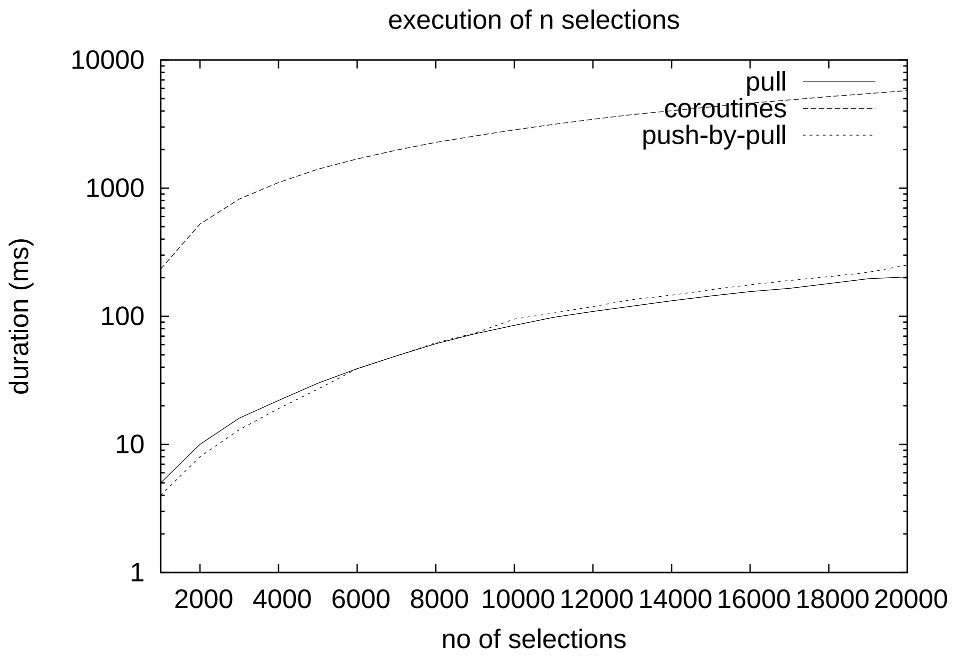

Figure 10.4.: A table scan with n selections

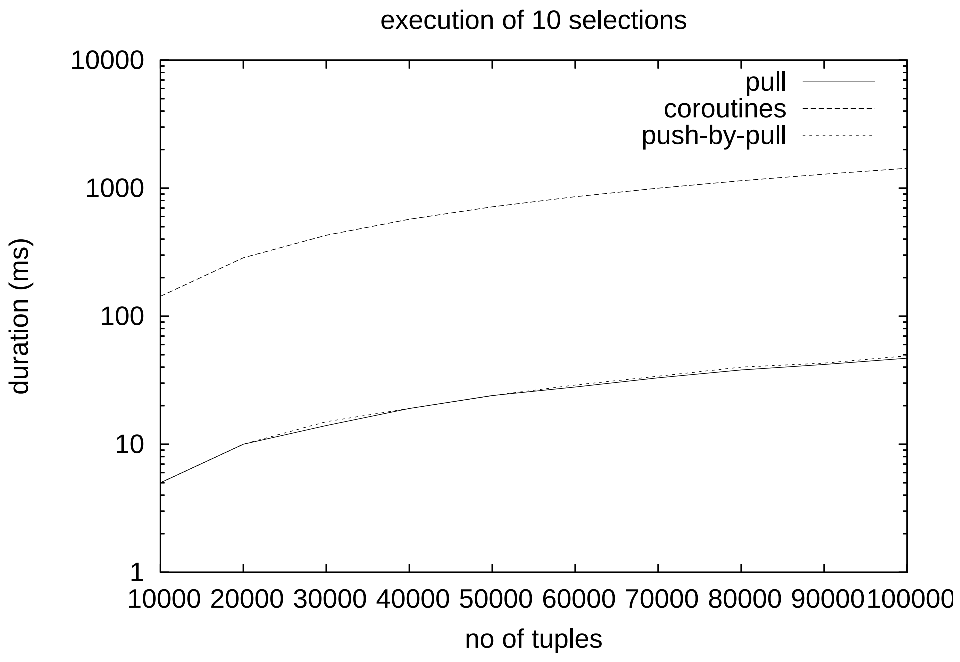

Figure 10.5.: A table scan with 10 selections

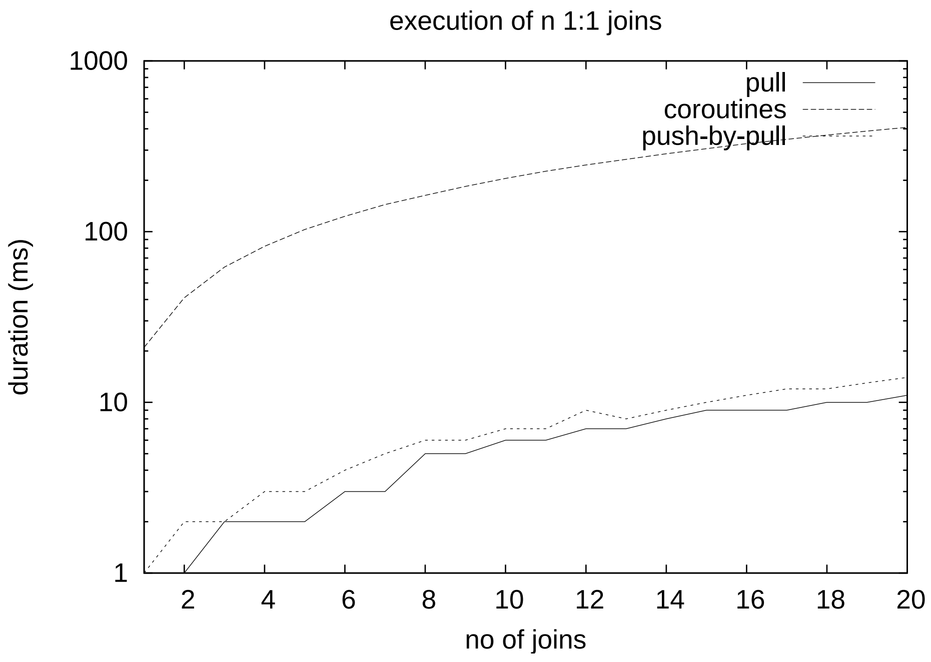

Figure 10.6.: A left deep query plan with n joins

by-pull approach is never invoked. To look at a more complex control flow, we constructed left-deep chains of nested loop joins, joining $n+1$ relations with 100 tuples each. For each output tuple each join operator must read from both its input streams, so that the control flow changes a lot. The operators are simple $1:1$ nested loop joins, so we mainly measure the overhead of the infrastructure.

The results are shown in Figure 10.6. As expected, the coroutine implementation is again very slow, as the operators are very simple. Besides, the push-by-pull implementation is somewhat slower, as it now has to perform more complex changes of control flow. But still the difference between the push-by-pull implementation and the pull implementation is at most 30%, and this is the worst case scenario: Every tuple requires $2n$ re-schedules (as each join has to switch from left to right and, for the next tuple, back) and the operators themselves require nearly no time (each join does just one integer comparison), so that only the cost for scheduling and data passing is measured.

Overall, the push-by-pull implementation performs reasonably well. The overhead involved is not too big and becomes even neglectible when more expensive operators or complex predicates are involved. As this approach allows the execution of far more general plans, the overhead can be justified easily.

10.7. Outlook

The experimental results show that DAG-structured query plans can be supported with a modest overhead. If necessary, this overhead could be reduced even more by using block oriented data passing [63].

Future work should try to improve the operators to handle arbitrary plans, as discussed in Section 10.5.4. In some cases, this could be done by reorganizing binary operators so that they accept input in arbitrary order (with respect to left and right). For some operators like a grace hash join, it can be done quite easily. Other operators like a nested loop join can be adapted with some work. In fact, for the problematic case (reading from the same input twice) a much more efficient implementation could be used: For a 1:1 join the implementation is trivial, and even for n:m the operator only has to check for boundaries between join attributes. Therefore, a database system could not only eliminate the restrictions for self joins but actually benefit from self joins.

11. Evaluation

11.1. Overview

In the previous chapters, we discussed several aspects of optimizing DAG-structured query graphs. However, we still have to show two claims: 1) creating DAG-structured query plans is actually beneficial and 2) situations where DAGs are beneficial are common and not constructed. Therefore, we present several queries for which we create tree-structured and DAG-structured query plans. Both the compile time and the runtime of the resulting plans are compared to see if the overhead for DAGs is worthwhile. All experiments were executed on a 2.2 GHz Athlon64 system running Windows XP. The plans were executed using the runtime system of the SOD2 object-oriented database system [58].

To avoid changing too many parameters at once, each operator (join, group-by etc.) is given 1MB of memory as buffer space. This is somewhat unfair against the DAG-structured query plans, as they need fewer operators and, therefore, could allocate larger buffers. But dynamic buffer sizes would affect the cost model, and the space allocation should probably be a plan generator decision. As this is beyond the scope of this work, we just use a static buffer size here.

Both tree-structured and DAG-structured query plans are constructed using the plan generator presented here, either with or without rules for DAG generation (i.e. information about share equivalence). While this allows a better comparison of the generated plan and the plan generation effort, this comparison is not completely fair, as tree structured query plans could be constructed using a simpler (and potentially faster) plan generator. However, the query execution time clearly dominates the query compilation time. Therefore, the resulting plan is more interesting than the compilation time.

11.2. TPC-H

The TPC-H benchmark [76] is a standard benchmark to evaluate relational database systems. It tests ad hoc queries where the database system must not be tuned for the expected queries (in contrast to TPC-R [75]). This results in query execution plans that are relatively simple and allow a better comparison between tree and DAG versions.

The schema is shown in Figure 11.1. It models a business database with customers, orders and supplier; the corresponding queries are from a data warehouse scenario. For the runtime evaluation we used the scale factor 1 database (1GB).

We now look at some exemplary queries. Note that queries without sharing

customer	lineitem	order	part
c_custkey	l_orderkey	o_orderkey	p_partkey
c_name	l_partkey	o_custkey	p_name
c_address	l_suppkey	o_orderstatus	p_mfgr
c_nationkey	l_linenumber	o_totalprice	p_brand
c_phone	l_quantity	o_orderdate	p_type
c_acctabl	l_extendedprice	o_orderpriority	p_size
c_mktsegment	l_discount	o_clerk	p_container
c_comment	l_tax	o_shippriority	p_retailprice
	l_returnflag	o_comment	p_comment
	l_linestatus		
	l_shipdate		
	l_commitdate		
	l_recepitdate		
	l_shipstruct		
	l_shipmode		
	l_comment		
150000 tuples	6001215 tuples	1500000 tuples	200000 tuples

supplier	partsupp	region	nation
s_suppkey	ps_partkey	r_regionkey	n_nationkey
s_name	ps_suppkey	r_name	n_name
s_address	ps_availqty	r_comment	n_regionkey
s_nationkey	ps_supplycosts		n_comment
s_phone	ps_comment		
s_acctbal			
s_comment			
10000 tuples	800000 tuples	5 tuples	25 tuples

Figure 11.1.: TPC-H Schema

```
select   ps_partkey, sum(ps_supplycost * ps_availqty) as value
from     partsupp,supplier,nation
where    ps_suppkey = s_suppkey and
         s_nationkey = n_nationkey and
         n_name = "GERMANY"
group by ps_partkey
having sum(ps_supplycost * ps_availqty) >
   (select sum(ps_supplycost * ps_availqty) * 0.0001
    from   partsupp, supplier, nation
    where  ps_suppkey = s_suppkey and
           s_nationkey = n_nationkey and
           n_name = "GERMANY")
order by value desc;
```

Figure 11.2.: SQL formulation of TPC-H Query 11

opportunities are unaffected by DAG support: The plan generator produces exactly the same plans with and without DAG support, and also the compile time is identical in our test scenario. Therefore, it is sufficient to look at queries which potentially benefit from DAGs.

Query 11

Query 11 is a typical query that benefits from DAG-structured query plans. It determines the most important subset of suppliers' stock in a given country (Germany in the reference query). The SQL formulation is shown in Figure 11.2. The available stock is determined by joining `partsupp`, `supplier` and `nation`. As the top fraction is requested, this join is performed twice, once to get the total sum and once to compare each part with the sum. When constructing a DAG, this duplicate work can be avoided. The compile time and runtime characteristics are shown below:

	tree	DAG
compilation [ms]	10.5	10.6
execution [ms]	4793	2436

While the compile time is slightly higher when considering DAGs (profiling showed this is due to the checks for share equivalence), the runtime is much smaller. The corresponding plans are shown in Figure 11.3: In the tree version, the relations `partsupp`, `supplier` and `nation` are joined twice, once to get the total sum and once to get the sum for each part. In the DAG version, this work can be shared, which nearly halves the execution time.

Query 2

Query 2 selects the supplier with the minimal supply costs within a given region. Structurally this query is similar to Query 11, as it performs a large join twice, once for the result and once to get the minimum (see Figure 11.4 for a SQL representation). However, it is more complex, as the nested query depends

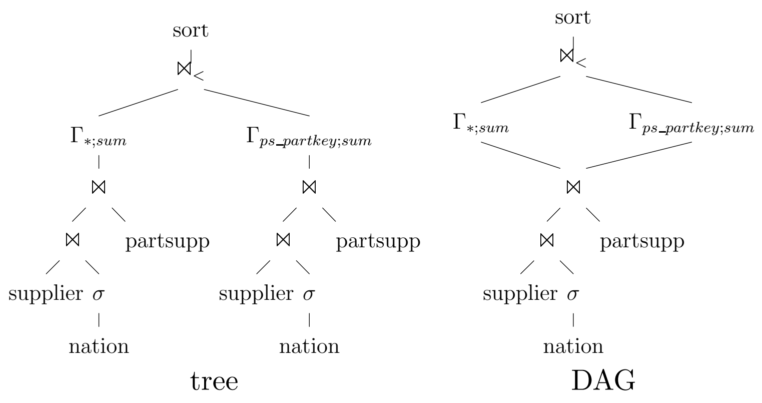

Figure 11.3.: Execution plans for Figure 11.2

```
select s_acctbal, s_name, n_name, p_partkey,
       p_mfgr, s_address, s_phone, s_comment
from   part, supplier, partsupp, nation, region
where  p_partkey = ps_partkey and
       s_suppkey = ps_suppkey and
       p_size = 15 and
       p_type like "%BRASS" and
       s_nationkey = n_nationkey and
       n_regionkey = r_regionkey and
       r_name = "EUROPE" and
       ps_supplycost = (
          select min(ps_supplycost)
          from   partsupp, supplier, nation, region
          where  p_partkey = ps_partkey and
                 s_suppkey = ps_suppkey and
                 s_nationkey = n_nationkey and
                 n_regionkey = r_regionkey and
                 r_name = 'EUROPE')
order by s_acctbal desc, n_name, s_name, p_partkey
```

Figure 11.4.: SQL formulation of TPC-H Query 2

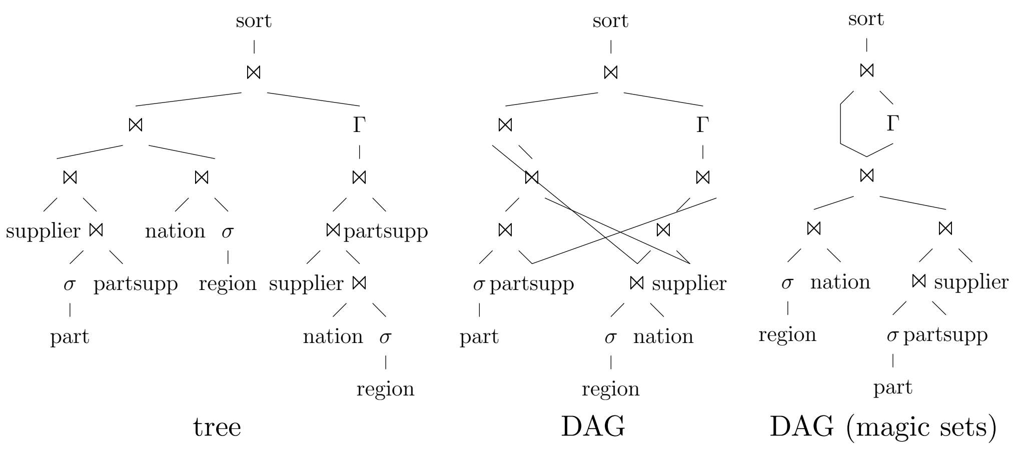

Figure 11.5.: Execution plans for Figure 11.4

on the outer query. We assume that the rewrite step unnests the query (by grouping the nested query on `ps_partkey` and using a join), but still the nested query lacks the `part` relation, which prevents sharing the whole join. Note that the relation (and the corresponding predicates) can be re-added by a magic set like transformation: The join with `part` is effectively a filter, as the join is a key/foreign key join. The nested query is joined by `ps_partkey` and the group-by is on `ps_partkey` and `part` is joined by `ps_partkey`, so the join can be duplicated inside the group-by without changing the result. Here, we consider three plan generation alternatives: Normal tree construction, DAG construction and DAG construction with rules for magic set transformation enabled. The runtime and compile time are shown below.

	tree	DAG	DAG (magic set)
compilation [ms]	9.3	9.2	9.7
execution [ms]	11933	7480	3535

The compile times for tree and DAG are about the same (the DAG is slightly faster, as it can ignore some dominated alternatives), while the magic set variant is about 5% slower due to the increased search space. The runtime behavior of the alternatives is very different, see Figure 11.5 for the corresponding execution plans. The tree variant simply calculates the outer query and the nested query independently and joins the result. The DAG variant tries to reuse some intermediate results (reducing the runtime by 37%), but still performs most of the joins in both parts, as the queries are not identical. When using the magic set transformation, large parts of the query become equivalent, which results in much greater sharing and also reduced aggregation effort, reducing the runtime by 70% compared to the tree variant.

11.3. Examples

Besides standard TPC-H queries, we also examine some queries selected to demonstrate certain optimization techniques. Note that these examples were

```
select *
from   order
where  o_orderstatus='F' or
       exists(select *
              from   lineitem
              where  l_linestatus='F' and
                     l_orderkey=o_orderkey)
```

Figure 11.6.: Disjunctive query

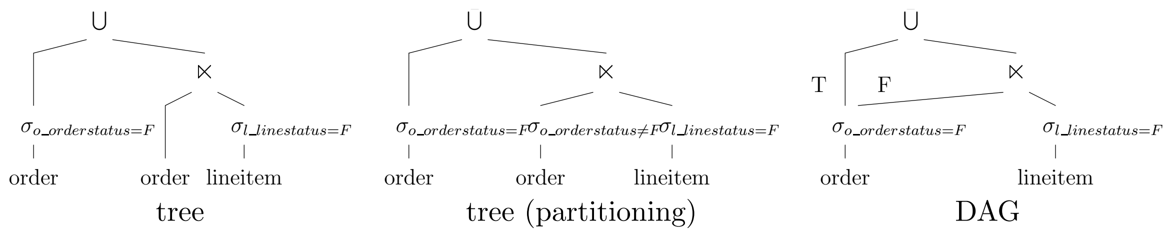
∪
⋉
$\sigma_{o\_orderstatus=F}$
$\sigma_{l\_linestatus=F}$
order
order
lineitem
tree
∪
⋉
$\sigma_{o\_orderstatus=F}$
$\sigma_{o\_orderstatus\neq F}$
$\sigma_{l\_linestatus=F}$
order
order
lineitem
tree (partitioning)
∪
T
F
⋉
$\sigma_{o\_orderstatus=F}$
$\sigma_{l\_linestatus=F}$
order
lineitem
DAG

Figure 11.7.: Execution plans for Figure 11.6

specifically chosen as a demonstration and are not necessary useful (e.g. the first query can be formulated much simpler by using schema information). However, they give an idea which optimizations are possible.

First, we consider a disjunctive query. We want to find all orders that are either finished or of which at least one item is finished. The SQL representation is shown in Figure 11.6. Note that for this very simple query plan generation is basically pointless (compile time $< 0.1ms$), as few decisions are possible, the main work is done during query rewrite. The compile time and runtime of three different alternatives are shown below, we will discuss the alternatives below.

	tree	tree (partitioning)	DAG
compilation [ms]	< 0.1	< 0.1	< 0.1
execution [ms]	46683	25273	22450

The direct translation of the SQL query into an execution plan would require a dependent join. We did not consider this alternative here, as a dependent join of `order` and `lineitem` would be prohibitively expensive. Instead, the `exists` expression is unnested and converted into a semi-join. As the other part of the disjunctive condition has to be checked also, the parts are evaluated independently and the result is combined. The resulting plan is shown on the left-hand side of Figure 11.6. The approach has two disadvantages: First, it requires an expensive duplicate elimination and second, it performs the join for tuples that already qualified. Both problems can be avoided by negating the first condition in the second branch, which guarantees non-overlapping results (second column of Figure 11.6). This transformation greatly reduces the runtime of the query, but is not trivial to do for all queries: Consider a query with two disjunctive `exists` conditions. In this case, negating the condition in the second branch would be prohibitively expensive. A more flexible approach is to use bypass plans which evaluate the first condition, return all qualifying tuples as result and pass only the other tuples to the second part of evaluation plan (right-hand side if Figure 11.6). This is even faster than the second approach and can be used efficiently even for very expensive conditions.

11.4. Conclusion

The experiments have shown that the compile time is mainly unaffected by DAG support when just considering sharing intermediate results. The compile time is affected when using new optimization techniques (e.g. magic sets in Query 2), but of course the search space is much larger there. Still the compile time is small and dominated by the runtime. The runtime effect of DAG support is very large, as sharing can drastically reduce the runtime of many queries.

In fact DAGs can be considered a clear win over tree-structured query plans. The compile time costs are minimal, the resulting plans are never worse than tree-structured plans and often the plans are much better than the equivalent tree-structured plans.

12. Outlook

The work presented here offers a solid base for creating and executing DAG-structured query plans. We presented a formalism to model DAG creation as a dynamic programming (respective memoization) problem and presented a plan generator that uses this formlism to create DAGs with little or no overhead compared to creating trees. We also discussed and solved the problems arising for the cost model and presented a runtime system suited for DAG-structured query plans that also handles simple trees with neglectible overhead. Finally, we have shown that DAG-structured query plans provide a great runtime benefit for real-life queries. Besides, we have shown examples where DAGs allow new classes of optimizations without buffering and without relying on multiple optimization phases. Overall, this shows that efficient DAG creation is possible and beneficial.

Future work should cover several topics: First the algebraic equivalences could be improved. While we have lifted the normal tree equivalences to DAGs, DAGs allow more transformations than trees. A theoretical foundation for this would be useful. Second, the current approach supports a wide range of optimizations, but relies on very smart rewrite and prepare phases to identify these opportunities. Either this should be formalized or the operator rules should be expanded to identify some of these opportunities themselves during plan generation. Related to this, some applications (XML processing and especially streaming) should be considered in more detail, as they can benefit greatly from DAG support. This involves both optimization rules and specialized operators.

Summarizing, DAGs offer great advantages for many real-life problems and allow an efficient implementation of interesting optimization techniques. Therefore, it is desirable to make DAGs the standard plan representation for database management systems.

A. Algebra

In this work, we used a number of algebraic operators. Although they are well known and described in literature [18, 52, 55], we provide definitions here for the sake of completeness. Similar to the relational model, we assume that each algebra expression produces a set of tuples with identical schema. We write $\mathcal{A}(e)$ for the set of attributes produced by an expression e (i.e. the attributes contained in each tuple) and $\mathcal{F}(e)$ for the set of free variables of the expression e. We write $a : b$ to assign the name a to the value b (e.g. when creating a new attribute a).

The most basic operator is a scan over a relation or an extent, but it does not belong to the logical algebra described here (as it is a physical operator). We just write R to get all tuples contained in relation R.

As algebra expressions produce sets of tuples, we can use the regular set operations $\cup|$, $\cap$ and $\setminus$ to construct new expressions. Note that that the $\cup$ operator has to perform duplicate elimination, as it produces a set. If it is clear that no duplicates can occur or if duplicates are relevant (i.e. for multi-sets), we write $\bar{\cup}$ for a union without duplicate elimination.

$$\begin{aligned}
e_1 \cup e_2 &= \{x | x \in e_1 \vee x \in e_2\} \\
e_1 \cap e_2 &= \{x | x \in e_1 \wedge x \in e_2\} \\
e_1 \setminus e_2 &= \{x | x \in e_1 \wedge x \notin e_2\}
\end{aligned}$$

Attributes are removed using the projection $\Pi_A(e)$ ($A \subseteq \mathcal{A}(e)$), renamed using $\rho_{a \to b}(e)$ ($a \in \mathcal{A}(e), b \notin \mathcal{A}(e)$) and new attributes are created (calculated) using the map operator $\chi_{a:f}$ ($a \notin \mathcal{A}(e), \mathcal{F}(f) \subseteq \mathcal{A}(e)$

$$\begin{aligned}
\Pi_A(e) &= \{\circ_{a \in A}(a : x.a) | x \in e\} \\
\rho_{a \to b} &= \{x \circ (b : x.a) \setminus (a : x.a) | x \in e\} \\
\chi_{a:f} &= \{x \circ (a : f(x)) | x \in e\}
\end{aligned}$$

A selection using a predicate p is written as $\sigma_p(e)$ ($\mathcal{F}(p) \subseteq \mathcal{A}(e)$). The cross product of two sets is written as $e_1 \times e_2$. If both operations are executed at the same time using a join, this is written as $e_1 \bowtie_p e_2$ ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$).

$$\begin{aligned}
\sigma_p(e) &= \{x | x \in e \wedge p(x)\} \\
e_1 \times e_2 &= \{x \circ y | x \in e_1 \wedge y \in e_2\} \\
e_1 \bowtie_p e_2 &= \{x \circ y | x \in e_1 \wedge y \in e_2 \wedge p(x \circ y)\}
\end{aligned}$$

Besides the normal join operator, there exist numerous special purpose join operators. The outer join $e_1 \bowtie_p e_2$ ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$) performs a join, but makes sure that every tuple from e_1 is part of the result. If no match was found, a match is constructed by setting the attributes from e_2 to `NULL`. The semijoin $e_1 \ltimes_p e_2$ ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$) only checks which tuples from e_1 match tuples from e_2, it does not construct the matches. Finally, the dependent join $e_1 \overrightarrow{\bowtie}_p e_2$ ($\mathcal{F}(p) \subseteq \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, $\mathcal{F}(e_2) \subset \mathcal{A}(e_1)$) performs a join where the evaluation of e_2 depends on e_1.

$$\begin{aligned}
e_1 \bowtie_p e_2 &= e_1 \bowtie e_2 \cup \{x \circ \circ_{a \in A}(a : \texttt{NULL}) | x \in e_1 \land \nexists y \in e_2 : p(x \circ y)\} \\
e_1 \ltimes_p e_2 &= \{x | x \in e_1 \land \exists y \in e_2 : p(x \circ y)\} \\
e_1 \overrightarrow{\bowtie}_p e_2 &= \{x \circ y | x \in e_1 \land y \in e_2(x) \land p(x \circ y)\}
\end{aligned}$$

The group-by operator $\Gamma_{A;a:f}(e)$ ($A \subseteq \mathcal{A}(e)$, $\mathcal{F}(f) \subset \mathcal{A}(e)$) builds groups of tuples with the same values in the group-by attributes and executes an aggregation function on the group. The unnest operator $\mu_{a:b}(e)$ ($b \in \mathcal{A}(e)$) converts one tuple with a set valued attribute into multiple tuples combined with the values contained in the attribute.

$$\begin{aligned}
\Gamma_{A;a:f}(e) &= \{x \circ (a : f(y)) | x \in \Pi_A(e) \land y = \{z | z \in e \land \forall a \in A : x.a = z.a\}\} \\
\mu_{a:b}(e) &= \{x \circ (a : y) | x \in e \land y \in x.b\}
\end{aligned}$$

Bibliography

[1] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst.*, 4(4):435–454, 1979.

[2] Sihem Amer-Yahia, Sophie Cluet, and Claude Delobel. Bulk-loading techniques for object databases and an application to relational data. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 534–545. Morgan Kaufmann, 1998.

[3] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[4] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 261–272. ACM, 2000.

[5] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts*, pages 1–16. ACM, 1986.

[6] D. Batory. Extensible cost models and query optimization in genesis. *IEEE Database Engineering*, 9(4):30–36, 1986.

[7] D. S. Batory. On the reusability of query optimization algorithms. *Inf. Sci.*, 49(1-3):177–202, 1989.

[8] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366. Morgan Kaufmann, 1994.

[9] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 1995.

[10] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[11] E. F. Codd. A database sublanguage founded on the relational calculus. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*, pages 35–68. ACM, 1971.

[12] A. Deshpande and J. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 30-September 3, 2004, Toronto, Canada*, pages 948–959. IEEE Computer Society, 2004.

[13] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[14] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 47–58. ACM Press, 1995.

[15] Johann Christoph Freytag and Nathan Goodman. On the translation of relational queries into iterative programs. *ACM Trans. Database Syst.*, 14(1):1–27, 1989.

[16] César A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–73, 1997.

[17] Sumit Ganguly, Akshay Goel, and Abraham Silberschatz. Efficient and acurate cost models for parallel query optimization. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada*, pages 172–181. ACM Press, 1996.

[18] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice-Hall, Inc., 1999.

[19] D. Gardy and C. Puech. On the effect of join operations on relation sizes. *ACM Transactions on Database Systems*, 14(4):574–603, 1989.

[20] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the ibm db2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[21] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 102–111. ACM Press, 1990.

[22] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[23] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[24] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[25] Goetz Graefe. The microsoft relational engine. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 160–161. IEEE Computer Society, 1996.

[26] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash joins and hash teams in microsoft sql server. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 86–97. Morgan Kaufmann, 1998.

[27] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 160–172. ACM Press, 1987.

[28] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

[29] Torsten Grust, Sherif Sakr, and Jens Teubner. Xquery on sql hosts. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 252–263. Morgan Kaufmann, 2004.

[30] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

[31] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM*

SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989, pages 377–388. ACM Press, 1989.

[32] Evan P. Harris and Kotagiri Ramamohanarao. Join algorithm costs revisited. *VLDB J.*, 5(1):64–84, 1996.

[33] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[34] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 656–667, Berlin, 2003.

[35] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[36] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 268–277. ACM Press, 1991.

[37] Yannis E. Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *TODS*, 18(4):709–748, 1993.

[38] Alfons Kemper and Guido Moerkotte. Access support in object bases. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 364–374. ACM Press, 1990.

[39] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. A blackboard architecture for query optimization in object bases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 543–554. Morgan Kaufmann, 1993.

[40] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[41] Tobias Kraft, Holger Schwarz, Ralf Rantzau, and Bernhard Mitschang. Coarse-grained optimization: Techniques for rewriting sql statement sequences. In *VLDB*, pages 488–499, 2003.

[42] Jürgen Krämer and Bernhard Seeger. Pipes: a public infrastructure for processing and exploring streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 925–926. ACM Press, 2004.

[43] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986.

[44] Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. Implementing an interpreter for functional rules in a query optimizer. In François Bancilhon and David J. DeWitt, editors, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, pages 218–229. Morgan Kaufmann, 1988.

[45] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 95–104. ACM Press, 1995.

[46] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107, 1994.

[47] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[48] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 18–27. ACM Press, 1988.

[49] Guy M. Lohman, C. Mohan, Laura M. Haas, Bruce G. Lindsay, Patricia G. Selinger, Paul F. Wilsm, and Dean Daniels. Query processing in r*. Research Report RJ4272, IBM Research Division, 1984.

[50] Raymond A. Lorie. Xrm - an extended (n-ary) relational memory. *IBM Research Report*, G320-2096, 1974.

[51] Raymond A. Lorie and Bradford W. Wade. The compilation of a high level data language. *IBM Research Report*, RJ2598, 1979.

[52] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[53] Robert Marek and Erhard Rahm. Tid hash joins. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, November 29 - December 2, 1994*, pages 42–49. ACM, 1994.

[54] William J. McKenna. *Efficient Search in Extensible Query Optimization: The Volcano Optimizer Generator.* PhD thesis, 1993.

[55] G. Moerkotte. *Konstruktion von Anfrageoptimierern für Objektbanken.* Informatik. Verlag Shaker, Aachen, 1995.

[56] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 247–258. ACM Press, 1990.

[57] M. Muralikrishna. Improved unnesting algorithms for join aggregate sql queries. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 91–102. Morgan Kaufmann, 1992.

[58] Thomas Neumann. Sod2 object-oriented database system, 2004. http://www.tneumann.de/sod2.

[59] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, 2005.

[60] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. Technical Report TR-03-011, Department for Mathematics and Computer Science, University of Mannheim, 2003.

[61] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 30-September 3, 2004, Toronto, Canada*, pages 960–971. IEEE Computer Society, 2004.

[62] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. In *Proceedings of the 20th International Conference on Data Engineering, 30 March - 2 April 2004, Boston, MA*, pages 461–472. IEEE Computer Society, 2004.

[63] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574. IEEE Computer Society, 2001.

[64] F. Palermo. A data base search problem. In *Proc. 4th Symp. on Computer and Information Sci.*, 1972.

[65] Jignesh M. Patel, Michael J. Carey, and Mary K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 56–66. ACM Press, 1994.

[66] Arnon Rosenthal and César A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In Hector Garcia-Molina and

H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 291–299. ACM Press, 1990.

[67] Prasan Roy. Optimization of dag-structured query evaluation plans.

[68] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 238–248. ACM Press, 1997.

[69] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[70] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 57–67. ACM Press, 1996.

[71] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[72] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.

[73] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing joins in disjunctive queries. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 228–238. Morgan Kaufmann, 1995.

[74] Arun N. Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 1994.

[75] Transaction Processing Performance Council, 777 N. First Street, Suite 600, San Jose, CA, USA. *TPC Benchmark R*, 1999. Revision 1.2.0. http://www.tpc.org.

[76] Transaction Processing Performance Council, 777 N. First Street, Suite 600, San Jose, CA, USA. *TPC Benchmark H*, 2003. Revision 2.1.0. http://www.tpc.org.

[77] Xiaoyu Wang and Mitch Cherniack. Avoiding sorting and grouping in processing queries. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany.* Morgan Kaufmann, 2003.

[78] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[79] Eugene Wong and Karel Youssefi. Decomposition - a strategy for query processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.

[80] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 89–100. IEEE Computer Society, 1994.

[81] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 345–357. Morgan Kaufmann, 1995.

[82] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4), Apr 77.