2
Approach

2.1 Specification

This chapter discusses the theoretical basis for the tool, starting with the specification based on the analysis of a test set of queries. Specifically, we will compile a list of all the requirements necessary to achieve a high SQL coverage as well as a high level of usability which will serve as the specification for the tool.

2.1.1 Test Set

In order to accurately establish the requirements for a general-purpose SQL query builder, we need to compile a set of queries covering as much as possible of the SQL language. At the same time, the set should be unbiased, putting the focus on actual real-life demands. To meet these requirements, the set was mostly assembled from queries taken from open-source applications. This turned out to be somewhat more involved than expected, as most larger applications make use of object-relational mapping libraries such as Red Hat's Hibernate, Java's own JDBC or ActiveRecord, which is part of the Ruby on Rails framework.
The test set can then be used to reveal strengths and weaknesses in the query builders discussed earlier and, in part, to verify that the tool developed in this thesis. Furthermore, since the queries represent a relatively unbiased selection, they can be used to get an idea of the relevancy of certain features by comparing their relative frequency.

2.1.2 Features

In order to achieve a high SQL coverage while providing a good user experience, a query builder must satisfy a number of features, which can be separated into three categories:

Relational Algebra containing the operators of the relational algebra, such as different types of joins, selection and projection.

SQL Features containing additional features of SQL that are not part of the relational algebra, such as aggregate functions.

UI & Usability containing features that improve the overall usablity, particularly for novice users, as well as characteristics of the graphical user interface.

The features of the first two categories are extracted directly from the definition of the relational algebra and SQL respectively. Since SQL implementations differ significantly between vendors, the features are relatively high level, focussing on semantic rather than syntactic constructs. For example, while the syntax to limit the number of entries in the result set varies greatly, the feature itself is present in all major implementations.
Since the features in these two categories represent elements of the specification of either the relational algebra or SQL, they directly influence the SQL coverage of the query builder. This is in contrast to features from the last category, which, from an coverage point of view, are non-essential. However, since the key purpose of query builders is to simplify the construction of queries, providing a high level of usablilty and a powerful user interface is just as important as achieving a high SQL coverage. The usability features were collected by analyzing existing query builders and concepts using queries from the test set and are therefore somewhat subjective. The list of requirements is presented in Table 2.1.

Relational Alg.	**SQL Features**	**UI & Usability**
Projection	Expressions	Visual
Selection	Sort	Abstraction
Rename	Limit	Contextual
Natural Join	Distinct	Correctness
θ-Join		Readability
Outer Join		
Semi/Anti-Join		
Divison		
Aggregation		
Subqueries		
Set Operations		

Table 2.1: List of features

Some of the features in this list, particularly the ones in the usability category, are somewhat non-descriptive and should therefore be explained.

Abstraction Does the user have to understand SQL to write a query?

Contextual Does the builder use available information, e.g. is it type-aware?

Correctness Are the generated queries syntactically correct?

Readability Are the generated queries readable?

Note that optimization of the queries is not part of this list. This is due to the fact, that performance varies greatly between vendors and is therefore difficult to assess without large scale testing. Also, the focus of this thesis is on the visual aspect of the builder rather than on the performance aspect.
The list of features can now be used to better understand the limitations of existing query buildes found in the introduction by evaluating, for every feature, whether it is supported by the query builders or not. Features that are only partially supported or not supported at all, but would be simple to implement (i.e. without significant changes to the user interface or the underlying architecture) are denoted as such.
From the results, we should be able to narrow down the most important areas that need to be covered in the specification. Furthermore, correlations between design choices and the impacts of these choices on the query builders will deliver valuable insights that will help in the design phase later on. The three query builders tested are the ones introduced in the introduction, EasyQuery, ExtJS and DFQL. All three are based on different interface concepts, leading to substantial differences in the test results which listed in Table 2.2.

	EasyQuery	ExtJS	DFQL
Relational Algebra			
Projection	yes	partial	yes
Selection	yes	yes	yes
Rename	yes	yes	yes
Natural Join	implicit	no	possible
θ-Join	implicit	partial	yes
Outer Join	implicit	yes	possible
Semi/Anti-Join	no	no	possible
Divison	no	no	possible
Aggregation	yes	yes	yes
Subqueries	partial	no	yes
Set Operations	no	no	yes
SQL Features			
Expressions	no	yes	no
Sort	yes	yes	yes
Limit	no	no	possible
Distinct	no	no	
Usability			
Visual	low	medium	high
Abstraction	medium	low	high
Contextual	high	low	low
Correctness	high	low	low
Readability	medium	high	

Table 2.2: Evaluation of the query builders

Green boxes indicate a positive test result, e.g. that the feature is fully implemented, while red boxes indicate the opposite, e.g. features that are not present at all or only in a very limited fashion. Finally, orange boxes indicate features that are only partially supported. Feature that are not implemented but could be added in a very straightforward way are also indicated by orange boxes. Since DFQL is not an SQL query builders, the readability of its output cannot be compared and is indicated by a gray box. Similarly, the distinct feature of SQL has no counterpart in DFQL, as it operates in the relational algebra.

In order to learn from the designs we studied, we need to analyze what characteristic of a query builder enables it to meet a certain requirement. As a consequence, the most interesting features are the ones, where some query builders outperform the others. If a feature is present in all three query builders, then we assume, that it can be implemented regardless of the interface concept chosen. Conversely, if a feature is not implemented in any of the query builders, we assume that it is hard, regardless of concept. Those features will have to be addressed independently, as there is no previous work to refer to.
DFQL significantly outperforms the other query builders in the first category. For the most part, it owes this to its dataflow based architecture. For instance, while DFQL implements neither the semi- nor the anti-join, both could be added with minial effort by simply adding two new nodes that cover those features. EasyQuery on the other cannot be extended in the same manner, as it handles joins implicitly. Similarly, ExtJS cannot be extended to support the feature, as joins are created by connecting two columns, creating the join condition for a θ-join.
Unlike the other two query builders, ExtJS does not allow for the same column to be projected multiple times. While one could argue, that this feature is very rarely used, it is essential for a large number of reasonable queries. For instance, it is required to construct a query that retrieves the maxiumum, minimum and average of a certain column.
Finally, two major differences in this category are found in the last two features. Once again, DFQL outperforms both EasyQuery and ExtJS by featuring full and arbitrarily deep subquery support as well as set operations. Both features are possible due to certain characteristics of dataflow programming that will be discussed later. For now, it suffices to say, that the concepts used by EasyQuery and ExtJS are static in term of control flow, fixed on a single query. Dataflow programming on the other hand introduces input dependencies (i.e. subqueries) and therefore supports user-defined control flow.

The second category is interesting, in that no query builder manages to achieve a satisfying coverage. Despite limiting itself to the relational algebra, DFQL offers additional features such as sorting and while it is not implemented as a primitive, a means to limit the number of output rows could be easily added as well. Both features, sort and limit, are not part of the relational algebra, which is based on sets that are unordered by definiton. For the same reasons, all rows in DFQL are distinct by definition, eliminating the need for a distinct modifier.
ExtJS does not support limit or distinct, but features sorting as well as arbitrarily complex expressions. This sets it apart from the other to query builders which do not support expressions. However, the feature is implemented in a completely non-visual way and has severe consequences on the usability of the tool. For instance, enforcing correctness on

textual input quite costly and might not be possible at all in certain cases.

Similar to the previous category, the results from the usability category reveal significant differences between the three tools. DFQL features a highly visual interface that covers most of its functionality while the visual part of ExtJS is limited to creating joins. As mentioned in the introduction, EasyQuery's interface has no visual elements at all, relying on a list based approach instead.
In order to simplify query construction for novice users, a visual query builder should abstract SQL into a more consistent form. For instance, SQL requires, that constraints on aggregates are placed in the HAVING clause as opposed to the WHERE clause. While this makes perfect sense from a technical point of view and is no issue to the experienced SQL user, novice users will likely find this behavior confusing. Therefore, it makes sense to abstract both types of constraints into a single construct. This particular problem is not addressed by any of the query builders, as neither ExtJS nor EasyQuery support conditions on aggregates and DFQL, being based on the relational algebra, never encounters the issue in the first place. In order to remove complexity, EasyQuery introduces the implicit join, based on an extended database schema. This works reasonably well but makes it impossible to construct cetain queries, e.g. queries that contain a self-join or a join that is not based on foreign keys.
Since the relational algebra is a mathematical concept and not a programming language, it is both concise and consistent. As DFQL merely provides primitves for a number of operators from the relation algebra, in inherits these properties. Combined with dataflow programming, this naturally leads to a very clean and uniform interface based on the simple concept of the node. Moreover, DFQL offers the ability to create subnodes, allowing the creation of new operators by combining primitives or other subnodes. This configurability makes DFQL highly abstract and sutiable for novice users as well as experts.

The key feature that sets EasyQuery apart from the DFQL and ExtJS is that it is highly contextual, using types and even table contents to limit the available options and provide hints to the user. For instance, EasyQuery adjusts the available comparison operators to the type of the column. Furthermore, aside from a few special cases, EasyQuery always produces syntactically correct queries. These two features are extremely valuable from a usability point of view as they greatly simplify query construction by providing hints to novice users, who are unable to understand and therefore manually debug the output. Neither of these two features are found in any of the other query builders. This is somewhat surprising, since support for the feature is not inherent to EasyQuery's architecture, and could therefore be implemented on the other tools as well. That being said, EasyQuery makes a number of compromises to simplify ensuring query correctness. Specifically, expressions based on textual input are difficult to check as discussed earlier and are therefore lacking.
While DFQL provides by far the most visual interface, many of its inputs, such as the join condition or the list of projected columns, are still text-based, therefore not making use of any contextual information.

In order to provide the best possible user experience, a query builder should output readable, concise SQL queries. This is not the case with EasyQuery, which doesn't use the star notation and instead defaults to projecting every column explicitly. Moreover, it automatically assigns an alias to every column, leading to massive SELECT statements. This is loosely related to correctness concerns that will be covered later in this thesis. For comparison, queries produced by ExtJS are a lot more readable, which, ironically, is due to the fact, that ExtJS doesn't spend any efforts processing the inputs. Finally, DFQL is meant as a replacement for SQL and consequently doesn't output any SQL code.

2.1.3 Discussion

A quick glance at the results suggests, that DFQL is more powerful than the other two query builders tested, and the feature-by-feature analysis delivers a signifcant amount of evidence to support this impression. DFQL not only provides a highly abstract and customizable interface but also manages to achieve the highest SQL coverage. While it has a number of limitations, mostly originating from its relatively heavy use of text-based inputs, these are not inherent the dataflow approach. Overall, the dataflow programming paradigm appears to be remarkably well suited to implement a visual SQL query builder.

2.2 Concepts

In this section, we will discuss some of the things we learned from the analysis in the previous section and how they influence the design of the tool. Specifically, we will study a number of correlations found in the test results and their implication on the design process.

2.2.1 Power and Dataflow

Traditional imperative languages model programs as a series of sequentially executed operations. Control flow is handled by means of conditional statements, based on the current state of the execution. Dataflow programming on the other hand focusses on the movement of data, treating functions as black boxes that are executed once all its inputs are valid. Therefore, a natural way to illustrate dataflow programs is to use graphs, where the nodes denote functions and the edges denote flow of data, connecting outputs of one node to inputs of another. While SQL was designed to be as similar to natural language as possible, graph representations are widely used to illustrate queries on the relational model. The reason this works particularly well, is that the relational model is an algebra and therefore adheres to the group axoims. Specifically, it satisfies the requirement of operational closure, meaning that the results of all operations on relations are also a relation. This is extremely beneficial when looking at graph-based languages, as it means that any output produced can be used as an input to any other node. With respect to database queries, this provides a very natural representation of nested queries, which is one of the most noted weaknesses of SQL. This explains, why DFQL performs so much better than query builders based on a more static approach. While DFQL only targets the relational algebra, we can assume that the benefits will also apply to a query builder covering a large part of all SQL statements.

2.2.2 Power and Correctness

The analysis of the test results revealed a strong negative correlation between the power of a query builder and its ability to produce correct queries. In other words, it appears that ensuring query correctness comes at the cost of significantly reduced power, as is the case with EasyQuery. This makes sense intuitively, as allowing a large number of queries to be constructed means that more cases need to be covered. For instance, supporting subqueries requires a very modular system capable of proving correctness for arbitrarily deep subquery chains.
While ExtJS stands out as the only query builder supporting expressions, it implements the feature using text-based inputs, sacrificing correctness in the process. DFQL shares similar weaknesses, as some of its inputs are text-based as well. This indicates, that in order to ensure that generated queries are syntactically correct, the tool should not rely on text-based input whenever possible.

2.2.3 Dataflow and Abstraction

One of the major strengths of dataflow programming is the concept of subnodes, essentially allowing the creation of new operators from primitives and other subnodes. In the case of DFQL, this results in a highly customizable and abstract interface.

2.3 Design

The design process is separated into multiple stages. We start out by deciding on a architecture and then from that derive the list of nodes. As a next step, we cover the functionality of the nodes one by one and finally cover some of the usability topics.

2.3.1 Architecture

As discussed earlier, dataflow programs can be represented as a graph, where nodes represent functions that exchange data across the edges. In the context of databases, the data could for instance be relations, columns or conditions. We will explore and compare two different approaches, the first being based on columns, the second being based on relations as their working units. Since the query is based on a graph, it is built piece by piece, it exists in multiple stages, meaning that the tool is unable to identify the query it is supposed to generate on its own. Both approaches therefore rely on an output node that transforms its inputs into the desired query.
The core idea of the column based approach is based on a table node that exposes all the attributes of the table, which can then be used directly to for instance add constraints or aggregate a certain attribute. Additionally, with the focus being on columns, projecting a column is This works quite well for simple queries such as aggregation and projection of a single column. This query is illustrated in Figure 2.1.
In the column based approach, most operations are implemented using the idea of pattern matching. In addition to the output, each column also has an input that can be used to con-

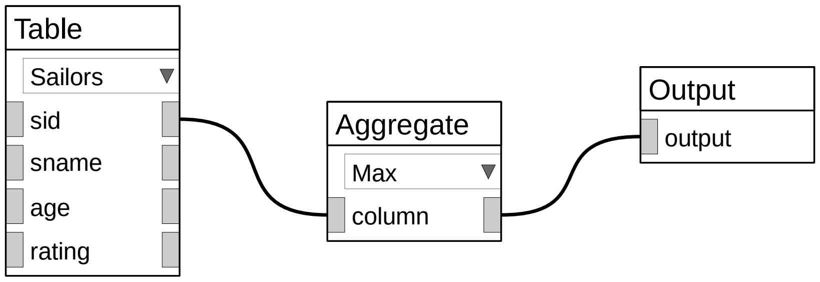


Figure 2.1: Simple query using column based approach

strain the values of the column. For instance, a query that returns all the ratings of sailors named Forrest is depicted in Figure 2.2.

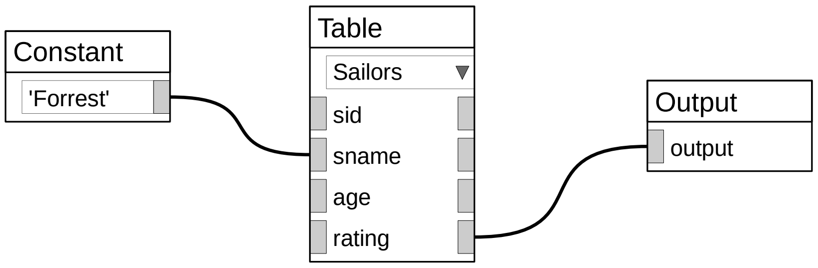


Figure 2.2: Pattern matching in the column based approach

Note that there is no limitation on the number of inputs. If multiple constraints are attached to input, all of them need to be met in order for a row to remain in the result set. For instance, to constraint the rating column to numbers between 5 and 10, we introduce a comparison node that takes a comparison operator and a constant and attach two of those, ≤ 10 and ≥ 5, to the input of the rating column.
Lists, including columns, can also be used as patterns as well, which are satisfied if the value occurs in the list. As such, list patterns are a simple implementation of SQL's IN-construct.
Overall, the pattern matching approach appears to work well for very simple queries but has a number of serious shortcomings. While list patterns can be used to emulate certain join types, they cannot modify the columns of the table and as such are unsuited to model the bevavior of a θ- or natural join. Therefore, an additional construct would have to be introduced to handle joins. Likewise, constraints on multiple columns, such as finding all the sailors with a rating that exceeds their age, are impossible to describe using this approach. There are numerous other features that exhibit similar problems, which could only be resolved by adding additional nodes, resulting in a two class system, where multiple ways can be used implement the same functionality. From a usability point of view, this is highly unfavorable and should be avoided whenever possible, as it is likely to confuse novice and expert users alike. Ultimately, a different, more uniform approach is required.
The second approach is based on relations and conceptually very similar to list processing in functional languages such as Haskell. It can be considered an extension of the relational algebra and inherits many of its beneficial characteristics, one of these being operational closure which was disscussed earlier. The core idea of the architecture is to shape the query in an

interative manner, with each node modifying the relation in a specific way. Metaphorically speaking, we can think of assemby lines that move data from worker to worker transforming it one step at a time. This works very well in concept, as relations can be understood as a list of tuples. For instance, a constraint on an column, like the one used in the query in Figure 2.2 could be implemented in Haskell using the filter function, which takes a list and a predicate and returns a new list consisting of all the tuples that satisfy the predicate. The same query implemented using the relation based approach is depicted in Figure 2.3.

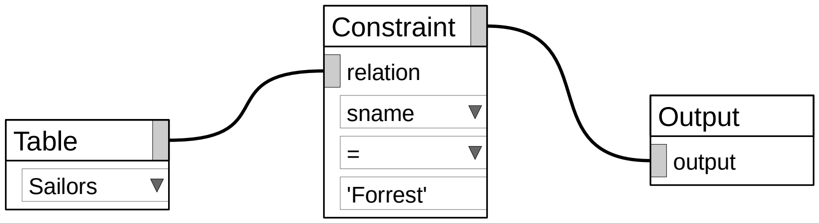


Figure 2.3: Selection in relation based approach

Note that since the data passed across the connection is a relation, it is now necessary, to declare what column is supposed to be constraint. In general, the approach feels somewhat less elegant than the pattern matching approach but can handle column-column constraints.
Due to the strong similarities between this architecture and the relational algebra, all relational operators can be translated directly. While this greatly simplifies the design process, it comes at the risk of remaining too conservative, resulting in a tool that does not represent a significant improvement over SQL.
An interesting aspect of the relation based approach is that the progressive construction of the query models the excecution of the query on the database. To support this intuition, a tool could in theory offer the possibility to inspect intermediate results, which could be tremendously helpful in debugging a query that doesn't return the expected results. DFQL actually implements this by allowing multiple DISPLAY nodes, that can be attached at any point in the query. In the case of DFQL, these intermediate are easy to obtain, since DFQL is not a query builder relying on a DBMS but actually executes the queries on its own. An SQL query builder could still emulate this behavior by running the intermediate queries. However, depending on the DBMS used and of course the query, this process could be very expensive, as the DBMS will most likely run the same queries multiple times. Additionally, it might use different execution plans for partial queries, resulting in inaccuaracies. Nonetheless, it could be a very useful feature and again underlines the advantages of dataflow based query builders.

2.3.2 Nodes

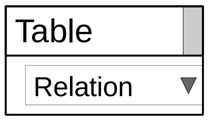
To get started, we compile a list of the nodes that we assume are necessary to cover most of the features in Table 2.1. Since this list of features is derived from the defintions of the relational algebra as well as SQL, most features will directly translate into their own node. Some straightforward simplifications, such as merging all join types into one node, will be applied without disscussion. The list is presented in Table 2.3. The names of the nodes are chosen to be more intuitive than the terminology used in SQL or the relational algebra.
As mentioned before, this list is based on the assumption, that the list of features translates

Node	Functionality
Table	Relation
Output	Output
Constraint	Selection (Filter)
Join	Joins
Select	Projection
Aggregate	Aggregation
Rename	Alias (Column)
Merge	Set Operations
Expression	Expressions
Sort	Sorting
Limit	Limiting

Table 2.3: List of Nodes

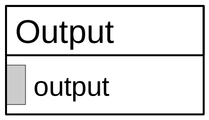
directly into the list of nodes. However, the discussion of the functionality of some of the nodes will reveal flaws leading to changes in the list.

Table



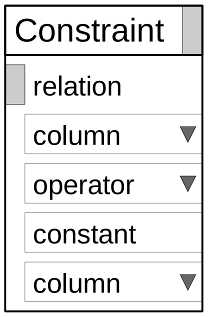
The Table node is one of the most basic nodes. It serves as the source of all the data in a query and as such doesn't have any inputs. The desired relation can be selected from a drop-down list. The SQL equivalent of the Table node is a relation in the FROM statement.

Output



The Output node serves as the "drain" of the query and is the only node without an output. Due to its unique role, it is also the only node that cannot be deleted and the only node present in a new, otherwise empty query.

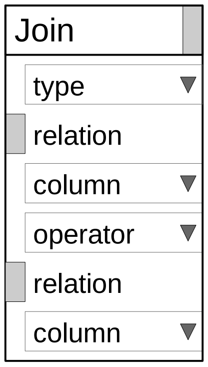
Constraint



The constraint node is used to remove rows from a relation that do not satisfy a certain predicate. Metaphorically speaking, it is a worker that filters all the rows with respect to the predicate. The SQL equivalent is the WHERE statement.
In DFQL, the predicate is supplied in text form. This represents a very flexible approach, as it allows arbitrarily complex expressions, but makes it difficult to ensure correctness. Furthermore, the user needs to be familiar with the syntax of the expressions, imposing difficulties for novice users. Therefore, we want to take a more user-friendly approach, by providing a fixed user interface that takes a column and

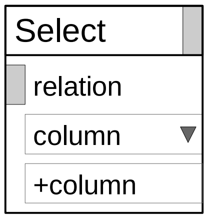
compares it to a constant or alternatively to another column. This approach also simplifies correctness evaluations, which will be discussed in a later section.

Join



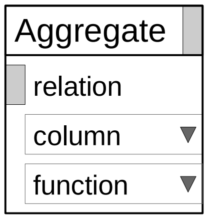
The join node, as the name suggests, handles all different types of joins. A drop-down list is used to indicate the type of the join. Depending on the type chosen, the node will change appearance, providing all the necessary inputs. A natural join for instance only requires two relations and the same holds for the semi- and anti-join. The θ-join on the other hand additionally requires a join condition. Therefore, the interface is expands, allowing the user to define a column to column comparison. The extended state, shown to the left, strongly resembles the constraint node. Furthermore, this representation does not allow complex join conditions but is instead limited to a single comparsion. Both facts will be addressed in the following section.

Select



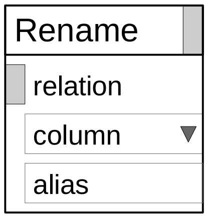
The select node is named after the SQL select statement and not the relational algebra operator and therefore represents a projection. Most query builders implement projection by providing a list displaying a checkbox for every column of the relation. This is approach is arguably intuitive but problematic, as it doesn't allow projecting the same column more than once. Also, reordering of the columns is impossible, a feature that is especially important in conjunction with set operators, as they require type-identical table layouts.
Due to these reasons, we examine a different, more flexible approach, featuring a dynamic column list. The user can attach a relation and then select a column from a drop-down list. Once a column is selected, another row is automatically added, meaning that an arbitrary number of columns can be selected.

Aggregate



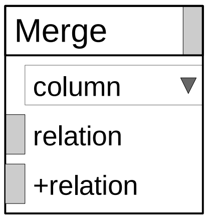
The aggregation node is relatively straightforward, allowing the selection of a column and aggregate function. However, it turns out that an implementation like this does not work in practice, since different behaviors might be expected depending on the type of query. Specifically, it is unclear whether the aggregation should replace the original column or not. This problem will be addressed in the following section.

Rename



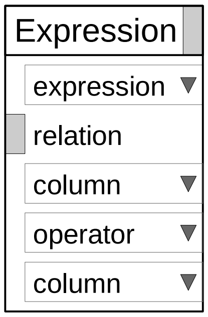
The rename node can be used to assign aliases to columns. The structure of the node is similar to the aggregation node, featuring a relation, column drop-down list and a text field to set the alias. Renaming is a somewhat problematic feature, due the fact that the column name must be unique, which can cause columns to be dropped upon joining tables. We will also discuss this in the next section.

Merge



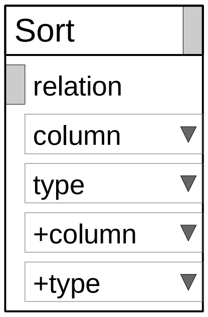
The merge node implements the set operators. Its interface works similarly to the projection in that it contains a dynamic list, taking an arbitrary number of relations. Set operations have special typing requirements on the columns of the relations. Therefore, they will be revisited in the section on correctness.

Expression



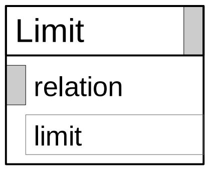
Due to the large number of different expressions, the interface of the expression node has to be completely dynamic. The structure shown to the left is tailored to allow arithmetic-like operations, for instance adding a constant to an integer column or a prefix to a text column. Other expressions such as acquiring the current system date, don't take any parameters while others might take many more. Additionally, expressions differ significantly between database vendors, meaning that the expressions would either have to be customizable or simply limited to a specific dialect.

Sort



The sorting node supports an arbitrary number of sorting attributes and as such requires a somewhat complex interface. The interface takes an arbitrary number of pairs of columns and sort types. Whenever a column is selected, a new one is appended to the list.

Limit



Finally, the limit node is very minimalistic and straightforward, taking only two parameters, a relation and an positive integer.

2.3.3 Issues and Refinements

This section discusses the various issues that became apparent in the previous section. We will address these issues by determining their root cause and then finding a way to resolve them. For reference, we will also examine how DFQL deals with these issues.

Logical Connectives

The assembly line metaphor works very well to illustrate the functionality of dataflow programs. Therefore, it can also be used to illustrate some of the weaknesses of the approach taken here. One major issues of the system arises from the fact that if a certain row fails to satisfy a constraint, the row is permanently removed. This, combined with the fact that the constraint node only allows the declaration of a single expression, means that disjunctions, OR statements in SQL, cannot be implemented at all. For instance, given predicates P and Q, and a row x in relation T where $P(x)$ and $\neg\, Q(x)$ hold, the composite predicate $P(x) \;\wedge\; Q(x)$ cannot be properly expressed. If Q is checked before P, x will be dropped and can't be recovered later on, because it is no longer in the relation. Likewise, a row y with $\neg\, P(y)$ and $Q(y)$ would be dropped if the order is reversed.
The only way disjunction can be implemented correctly in this approach is by splitting the base relation output, applying the constraints independently and then combining the results. This approach is illustrated in Figure 2.4.

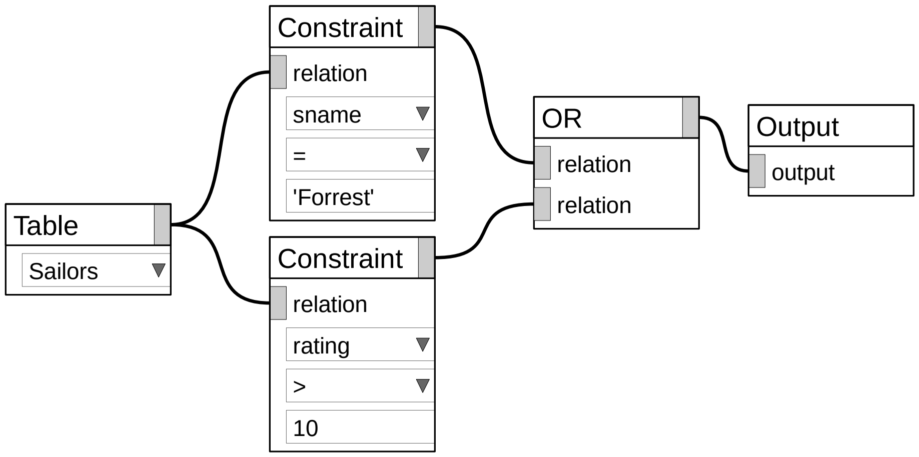


Figure 2.4: Disjunction (by split and merge)

The OR node used in this example is not defined yet. However, it turns out that it is semantically equivalent to a UNION node, i.e. a merge node set to union. Again using the assembly line metaphor, the union operation combines all rows coming from two different assembly lines onto one, while removing duplicates, which is equivalent to a single worker removing all rows that don't satisfy any of the predicates. The difference is that the resulting SQL query uses a UNION operation instead of an OR statement, which can be assumed to be far more expensive. Moreover, the order of the rows might differ, resulting in different results if limit is used in conjunction with aggregation. Nonetheless, we already have a working implementation that is semantically correct, but at the cost of performance and readability.
Note that conjunctions do not share the same issues. This is due to the fact that, in the metaphor, we simply have two workers independently removing rows according to their predicate. An example is shown in Figure 2.5.

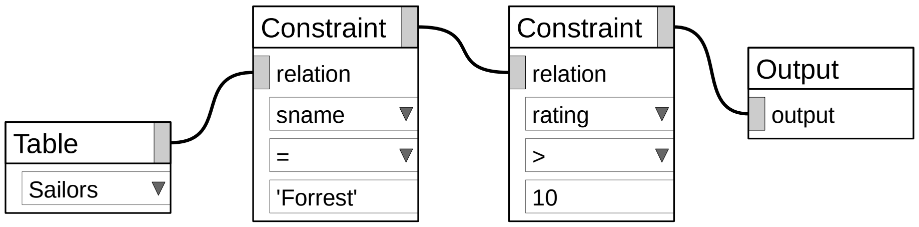


Figure 2.5: Conjunction (by linking)

DFQL deals with this issue by using text-based constraints. This allows for arbitrarily complex predicates, but as discussed earlier, significantly complicates correctness validations.

Joins

In the previous section, we designed the join node in such a way that it can handle all different types of joins. This is achieved by using a dynamic interface that adjusts the parameters depending on the type of join. Interestingly, the interface for the θ-join turns out to be almost exactly the same as the interface for the constraint node. This makes sense, if we think of the θ-join as simply a condition across multiple tables, which therefore requires a join. Accordingly, we can argue that instead of providing two mostly identical nodes, we should combine the functionality into one. Combining nodes is generally a good thing, as keeping the number of nodes low should, in theory, lead to a more user-friendly tool. On the other hand, the complexity of each individual node should be kept as low as possible and the behavior should be consistent and sensible. For instance, using a union operator to model a boolean disjunction is semantically correct, but not intuitively clear. In this case however, again using the intuition that a θ-join is simply a condition across tables, combining the two nodes does not obscure their functionality. However, the question remains whether the combination can be performed without significantly increasing the complexity.
Figure 2.6 shows how the two nodes could be combined. The new constraint node contains the same number of parameters as the join node and one more than the basic constraint node. Furthermore, in order to constrain two attributes of the same relation, we now need to attach the relation to both relation inputs. On the other hand, defining a join implicitly, based on a condition is potentially far more intuitive to users unfamiliar with the relational algebra and

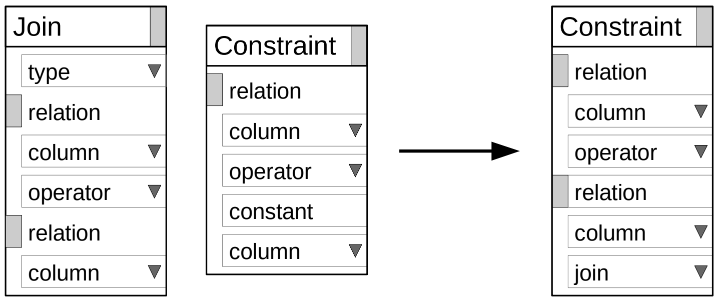


Figure 2.6: Combining the join and contraint nodes

therefore should have a positive impact on usability.
While EasyQuery also features an implicit join, there are a number of notable differences. Both query builders implicitly create a join for constraints across multiple tables. However, EasyQuery joins the tables based on foreign key relations and additional information provided with the schema, while the approach takes here simply joins the tables that are involved in the condition. This approach is a lot more flexible and grants the user more control over the construction of the query. For instance, it is powerful enough to express a self-join, something that cannot be modeled in EasyQuery. Additionally, the approach is also a lot more transparent, which is especially useful for more experienced users.
Up to now, the new node only properly covers the θ-join and we still have to address the natural, semi- and anti-join. All three remaining types join the tables based on attribute names and therefore don't need any additional inputs. This behavior cannot be properly expressed using the current layout of the new constraint node. Therefore, we could choose to keep a simplified version of the old join node or alternatively drop support for these types of joins altogether. An other idea involves constructing a new type of join that combines characteristics of both the θ- and semni-join. This join would take two relations and a condition and return a relation that only contains the columns from the first table and all the rows for which there is at least one row in the second table for which the condition is satisfied. For all intents and purposes, this is a semi-join that uses a condition instead of matching column names. In fact, this behavior might be preferable due to the fact that it gives the user more control over the join condition used. The anti-join can be implemented analogously.
The only type left is the natural join. However, semantically speaking, a natural join is simply a special case of an equijoin which is itself a special case of a θ-join. Additionally, the argument that joining based on column names leaves the user with little control applies to the natural join as well. In conclusion, keeping the join node for the sole purpose of supporting the natural join is not justified and we instead drop the feature.

Aggregation and Projection

In addition to the changes applies so far, we also combine the aggregation and projection nodes. Unlike the combination of the contraint and join nodes, which was based on usability concern and therefore non-essential, combining aggregation and projection is required in order to provide the full functionality.
The assembly line theorem we have used so far suggests, that all operations that are applied

to a relation take effect immediately. For instance, when a constraint is applied to a relation, the rows that do not satisfy the constraint are removed immediately. Likewise, an aggregation is applied directly, effectively changing the number of tuples in the relation. However, these two examples are treated differently by the query generator. The constraint results in a query, that can still be modified. For instance, two constraints applied one after the other will result in a single query, without any use of subqueries. Aggregation on the other hand "finalizes" a query, meaning that any subsequent nodes may not change it any way. For instance, a constraint on an aggregated query will create create a new query, using the aggregated one as a subquery. This is due to the fact, that while many of the relational operators are commutative, aggregation is not. For example, assume that we want to rename a column and apply a constraint to a relation. In that case, the execution order of the two operations will not change the outcome. However, if we instead aggregate the column and apply the constraint, the order does indeed matter. Therefore, the output produced by the aggregation node has to be treated as a subquery.
In order to understand, why this requires combining the aggregation and projection nodes, we examine the following simple query:

```
SELECT sname
FROM Sailors;
```

Now, we aggregate the query by applying COUNT to $sname$. This can result in the following two queries:

```
SELECT COUNT(sname)
FROM Sailors;
```

```
SELECT sname, COUNT(sname)
FROM Sailors
GROUP BY sname;
```

Both queries return completely different results. The first query returns the number of sailors stored in the table while the second one returns the number of occurrences of every name in the table. However, both queries are perfectly reasonable, yet there is no way for the user to influence which of the two is going to be generated, which would result in a considerable loss of power.
Combining aggregation and projection solves this issue, because it allows the user to exactly specify the desired query. In the example, since projection and aggregation happen simultaneously, the user can decide manually whether or not to include $sname$ in the projection. The layout of the new node is relatively straightforward and depicted in Figure 2.7.

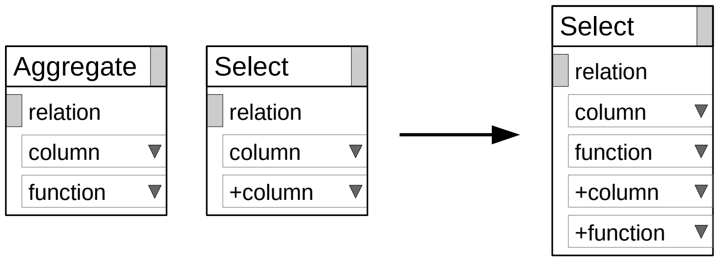


Figure 2.7: Combining the aggregation and projection nodes

DFQL handles this issue by adding an additional input to the aggregation node that takes a list of columns to be grouped which is quite similar to the approach taken here. However, since that node cannot be used to project columns without aggregating, DFQL retains a pure projection node.

Aliasing

Due to the fact that SQL uses column names as unique identifiers, aliasing can result in unexpected behavior. Consider the following query:

SELECT *
FROM Sailors s INNER JOIN Reserves r ON s.sid = r.sid;

Despite the fact that both tables contain a column named sid, the output relation only contains one such column. In this example, this is perfectly reasonable, due to the fact that the two columns are equijoined and therefore identical. That is not necessarily the case though. Self-joins in particular will often cause columns to be dropped and thus require extensive renaming. Interestingly, the fact that any data is lost is irrelevant to the query builder, as it is oblivious to the data anyway. However, in order to keep track of typing information, it needs to know which of the two columns is dropped. This is relevant for set operations, where the type of all columns is needs to match.
One way to completely avoid this problem is by making sure that no column will get dropped in the first place. A simple technique to achieve this, is to generate generic aliases for every column. While this approach is very effective, it results in bloated queries, due to the fact that every column will be projected separately. This significantly reduces query readability and should therefore be avoided. Instead, aliases should be only be generated when necessary, which requires keeping track of table names when joining.

2.3.4 Correctness

Scoping

Type Inference

2.3.5 Subnodes

2.3.6 Query Generation

Grammar