FluiDB: Adaptive storage layout using reversible relational operators

Christos Perivolaropoulos

Doctor of Philosophy

The University of Edinburgh

January 2022

To my dearly departed cat,

Roxy

Abstract

It is a popular practice to use materialized intermediate results to improve performance of RDBMSes. Work in this area has focused either on optimizers matching existing results or selecting useful intermediate results from a plan, but few attempts have been made to create plans with intermediate results in mind, and none that make any deduplicate the stored data to alleviate the storage cost of maintaining possibly large queries.

We built *FluiDB* to explore a novel approach to integrating the selection of materialized results with the planner in order to optimize the logical representation of data in memory. FluiDB materializes hot intermediate results and deduplucates data to alleviate the cost of maintaining them. This is achieved by introducing *reversible operations*, versions of normal relational operators that optionally pass enough data to the output to make the input relations reconstructable. A planner aware of such operations can build query plans that dynamically adapt the data layout to the plan under constrained memory budget. This thesis revolves around four main chapters each of which describes in detail a different part of FluiDB and a final one that goes into evaluation of the system.

The first chapter focuses on query processing and the relational algebra semantics that FluiDB operates under. FluiDB parses queries into graphs of sub-queries connected by reversible operators. Each such graph of the workload is merged into a global query graph that is used to infer properties of each relation like cardinality and extent.

The next chapter is dedicated to the planner and a novel monad for weighted back-tracking that the planer is based on. The planner attempts to generate a plan based on the query graph that besides solving the query at hand, leaves in memory an optimal set of queries for the workload being run. In this chapter, the garbage collector is also discussed, that creates deletion operators as part of the plan such that the available budget is respected.

After that, we go into *Antisthenis*, a novel incremental computation system used by the query planner to determine, given a particular set of materialized relations, whether a relation materializable and the expected cost of materializing a relation. Antisthenis, besides reusing computations, takes advantage of the properties of the operations involved in the expression she is evaluating, like absorbing group elements, to heuristically avoid as much work as possible.

The final chapter about the FluiDB architecture describes the transpilation of plans generated by the planner to C++, as well as the supporting libraries that enable the evaluation of queries as C++ code, and the low level data organization of the database. The thesis closes with a chapter that describes our methods for benchmarking and some experimental results.

Lay Summary

A lay summary.

Acknowledgements

Thank some people.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Christos Perivolaropoulos

Contents

List of Figures

CHAPTER 1

Introduction

Something about the beginning of a journey.

(Author)

Chapter summary

- FluiDB is an in-memory RDBMs optimizes data layout for space efficiency w.r.t. the workload
- The main novelty relates to the introduction of reversible relational operations which affords a new perspective on query planning and view selection.
- She materializes all intermediate nodes and deletes garbage collects when she runs out of space.

With the advent of technologies that make access to information scalable and affordable, the mental and temporal gap between collection of data and their analysis grows rapidly. At least two of the biggest players in the the tech industry, Facebook and Google, base their competitive advantage almost exclusively on vast amounts of information that they have collected and their capacity for such collection and processing. In cases like these the layout of the stored data is independent of the growing number of applications taking advantage of it.

A mantra in database community used to be that "storage is cheap" and while that is true a more complete version of the mantra might be that "slow storage is cheap".

Databases have traditionally been dealing with the tradeoff between memory and time efficiency within monetary constraints especially with the use of intermediate result recycling technologies which employ sophisticated ways of chosing parts of computation to be stored for reuse. This model has had some impressive results in OLAP workloads. We adopt a slightly different view of the problem: executing query plans does not only leave some specific intermediate results as residue, but rather *transitions the entire storage state from one where the result of the query is not materialized to one where it is*. Whith such a notion of query planning what new dimensions open up in the desgn space of a query planner? The creation of FluiDB is an attempt to study some aspects of this question. In particular FluiDB is based on the following pillars:

- Reversible query operations that allow for more sophisticated plans based on available materialized views.
- The planner involves a garbage collector that will delete materialized views or primary tables that can be materialized from the remaining relations.
- An incremental numeric evaluation system that allows the planner to efficiently and repeatedly infer the cost of materializing queries under a rapidly changing inventory of materialized views.
- Query execution driven based code generation.

These concepts allow FluiDB to dynamically adapt the data layout to the workload in ways that would not be possible in traditional intermediate view recycling systems. To make this more concrete let's look at an example.

Consider the following query over the TPC-H dataset that computes the average discount per country:

```sql
select      n_name, avg(l_discount)
from        lineitem, customer, nation, order
where       l_orderkey = o_orderkey
and         c_custkey = o_custkey
and         c_nationkey = n_nationkey
and         l_shipdate > 10-11-2015
group by    n_name
```

An optimizer considering this query in isolation would come up with some plan resembling the following following plan following:

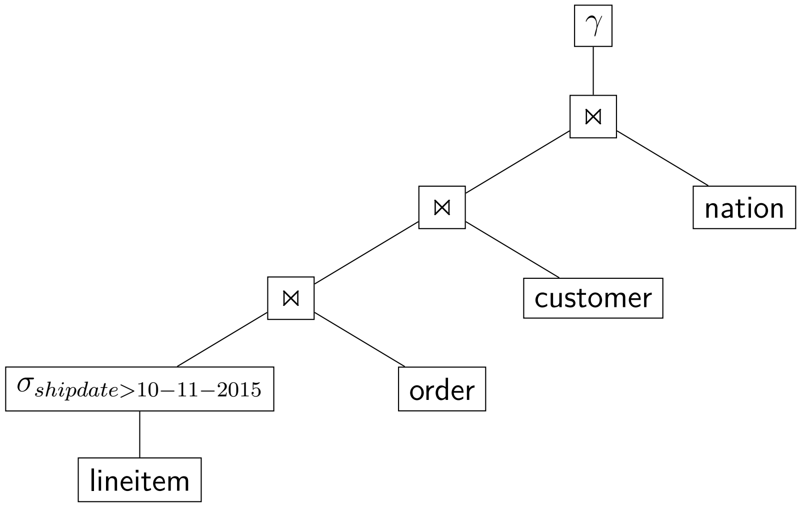


Figure 1.1: An efficient logical plan for a single query.

Notice how the selection ($\sigma_{shipdate>10-11-2015}$) is pushed all the way to the bottom of the tree because it is a cheap operation (worst case a single scan over the input) and can potentially shrink the input by a lot rendering the joins higher in the tree much cheaper.

Consider a workload that repeatedly runs a query generated from the following template:

```sql
select      n_name, avg(l_discount)
from        lineitem, customer, nation, order
where       l_orderkey = o_orderkey
and         c_custkey = o_custkey
and         c_nationkey = n_nationkey
and         l_shipdate > ${min_date}
group by    n_name
```

It is clear in this case that it would be beneficial if the workload is large enough for the cost of $lineitem \bowtie order \bowtie customer \bowtie nation$ to be amortized we would like to materialize the large join and only run $\gamma\sigma$ for each query attached as show in figure 1.2.

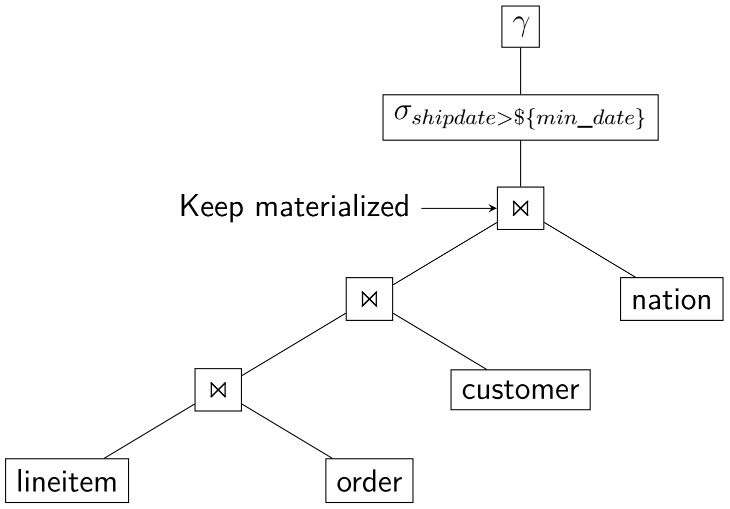


Figure 1.2: An efficient logical plan for a template based workload.

This shift of focus from per-query optimization to considering the amortized cost of materialization of expensive relations is already a tall order for most RDBMSes that feature materialized views. A simple but effective approach to integrating incremental query materialization and the optimization processes, was presented in [78]. In their approach they maintain a *history pool* (a list of all the past queries) that is used to decide the benefit of materializing a sub-expression, and a *view pool* that keeps track of the materialized tables at every moment. Both these sets are taken into account during planning to produce a plan that will likely minimize the amortized cost of the workload. After the query is planned the sets are updated. A limitation of such an approach is that when dealing with relations like $lineitem \bowtie order$ in budget restricted settings, materialized view storage can quickly become a scarce resource.

There is an opportunity to reduce the effect of this problem by exploiting another common workload attribute: certain tables are frequently subsumed by the same intermediate result. In our example workload $lineitem$ is fully subsumed by $lineitem \bowtie order$, $lineitem \bowtie order \bowtie customer$, and $lineitem \bowtie order \bowtie customer \bowtie nation$. Similarly

$opreder$, $customer$, and $nation$ are fully or partially subsumed by these relations. It seems wasteful to always keep the rows of all primary tables separately *and* concatenated in the rows $lineitem \bowtie order \bowtie customer \bowtie nation$.

In particular we can retrieve the rows of $lineitem$ by projecting on the latter relation and deduplicating the resulting rows we can obtain any of the former. In this – contrived yet demonstrative – example a plan taking into account amortized costs could be:

```
Query $\gamma\sigma_{p_0}(lineitem \bowtie order \bowtie customer \bowtie nation)$ {
  $Q_0$ := Materialize[$lineitem \bowtie order$]
  $Q_1$ := Materialize[$customer \bowtie Q_0$]
  $Q_2$ := Materialize[$nation \bowtie Q_1$]
  # Not enough space to continue. Delete relations that we can rebuild.
  GC {
    Delete[$lineitem$]
    Delete[$Q_0$]
  }
  $Q_3$ := Materialize[$\sigma_{p_0} Q_2$]
  $Q_4$ := Materialize[$\gamma Q_3$]
}
Query $\gamma\sigma_{p_1}(lineitem \bowtie order \bowtie customer \bowtie nation)$ {
  GC {
    Delete[$Q_1$]
  }
  $Q_5$ := Materialize[$\sigma_{p_1} Q_2$]
  $Q_6$ := Materialize[$\gamma Q_5$]
}
Query $\gamma\sigma_{p_2}(lineitem \bowtie order \bowtie customer \bowtie nation)$ {
   GC {
    Delete[$customer$]
   }
  $Q_7$ := Materialize[$\sigma_{p_2} Q_2$]
  $Q_8$ := Materialize[$\gamma Q_7$]
}
...
# Since the large join has all the columns of \li we should be able
# to create it by simply getting slicing and deduplicating
Query $lineitem$ {
  $lineitem$ := Materialize[$uniq\{\pi_{cols(lineitem)} Q_2\}$]
}
```

Figure 1.3: A sequence of plans optimizing the workload amortized cost and involving reverse operations.

By incorporating reverse relational operations where possible FluiDB can indeed generate workload plans similar to the one described.

The solution we experiment with by implementing FluiDB resembles the solution provided in [19] by Gou et.al for multi-quer. In their work they embed aggregations `group by x1, .., xk` into the $\subseteq$-lattice that arises from the powerset $P(\{x_1, ..., x_k\})$. Thereby they encode the fact that `group by A, B, C` is subsumed, or can be computed by, either of `group by A, B`, `group by A, C` or `group by B, C`. Once the lattice is constructed a variant of the $A^\star$ path finding is used algorithm to search for the optimal aggregation plan. However they make no attempt to recycle tables, ie. garbage collect tables, whose data can be found in other relations, and narrows it's attention to aggregations.

From the aforementioned work we keep the basic notion of using a graph to represent the subsumption of queries and to derive the benefit of materializing a relation. We also use path finding techniques in that graph to create plans. However we introduce a more complex and ad-hoc hierarchy of relations to account for subsumptions the entire relational algebra, rather than just aggregations that is very similar to AND-OR dags as found in [15]. Thereby we express for example the fact that $\sigma_p(S)$ subsumes $\sigma_{p\vee q}(S)$, or that $B \bowtie C$ subsumes $A \bowtie B \bowtie C$. Furthermore the relations we express in that graph are bidirectional. So rather than only finding paths towards the goal and deleting relations when they are no longer needed, we simultaneously plan for moving towards the goal query and performing "backwards" operations for saving up space. We clarify this with an example which demonstrates a slightly simplified version of our system's functionality.

In figure 1.4 we provide representation of a subsumption graph that our RDBMS might create after witnessing selections on $lineitem$. For brevity let $p := shipdate > 10 - 11 - 2015$, $q := quantity < 24$ and $r := discount < .06$. The RDBMS has encountered $\sigma_p(lineitem)$, $\sigma_q(lineitem)$ and $\sigma_q\sigma_r(lineitem)$.

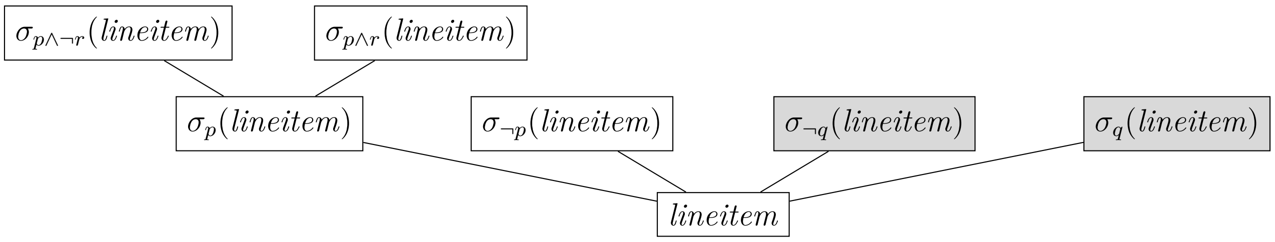

$\sigma_{p\wedge\neg r}(lineitem)$
$\sigma_{p\wedge r}(lineitem)$
$\sigma_p(lineitem)$
$\sigma_{\neg p}(lineitem)$
$\sigma_{\neg q}(lineitem)$
$\sigma_q(lineitem)$
$lineitem$

Figure 1.4: The materialized relations are marked with grey. Assuming the absense of null values and assuming set semantics $lineitem = \sigma_p lineitem \cup \sigma_{\neg p} lineitem$. FluiDB can find a plan to generate any of the relations in the graph from $\sigma_p lineitem$ and $\sigma_{\neg p} lineitem$.

The query that we are planning is $\sigma_{quantity<24}\sigma_{discount<.06}(lineitem)$, which is denoted in the figure as $\sigma_{q\wedge r}(lineitem)$. Our total size budget is 2.5.

Following the edges in the graph, to plan $\sigma_{q\wedge r}(lineitem)$ we need $\sigma_q(lineitem)$ and for that we need $lineitem$. So first the union:

$$\sigma_{\neg p}(lineitem) \cup \sigma_p(lineitem) \rightarrow lineitem$$

Then we need $\sigma_q(lineitem)$ but we are now using 2 units of space and adding .6 more would exceed our space budget of 2.5. $lineitem$ is the least beneficial of our materialized views but it is required for our next step, ie. creating $\sigma_q(lineitem)$. $\sigma_{\neg p}(lineitem)$ is deleted since it's derivable from $lineitem$ and is least beneficial. Then $\sigma_q(lineitem)$ is created and now we are using 2.1 units of space.

$$lineitem \rightarrow \sigma_q(lineitem)$$

Finally we need to create the final relation $\sigma_{q\wedge r}(lineitem)$. However it's space requirement is .5 and we we would be exceeding our budget. We can't delete $lineitem$ even though it is our least beneficial table because we would have no way of recreating it.

Here we have two options. One is to delete $\sigma_{\neg p}(lineitem)$, our most benefit al table. The other, which is the one that the system should opt for, is to backtrack. When we created $\sigma_q(lineitem)$, instead we create both $\sigma_q(lineitem)$ and $\sigma_{\neg q}(lineitem)$.

$$lineitem \rightarrow \{\sigma_q(lineitem), \sigma_{\neg q}(lineitem)\}$$

Once both those tables are materialized we can safely delete $lineitem$. Now our space usage is 1.1 and we can safely

$$\sigma_q(lineitem) \rightarrow \sigma_{q \wedge r}(lineitem)$$

Which is the requested query and, in summary, the final can be represented as:

```
Inventory {
  |\(Q_0\)| := |\(\sigma_{\neg p}(\li)\)|
  |\(Q_1\)| := |\(\sigma_{p}(\li)\)|
}
Query $\sigma_{q \wedge r} lineitem$ {
  $lineitem$ := Materialize[$Q_0 \cup Q_1$]
  GC { Delete[$Q_0$] }
  $Q_2$,$Q_3$ := Materialize[$\{\sigma_q(lineitem), \sigma_{\neg q}(lineitem)\}$]
  GC { Delete[$lineitem$] }
  $Q_4$ := Materialize[$\sigma_r Q_2$]
}
```

Figure 1.5: A sequence of plans optimizing the workload amortized cost and involving reverse operations.

This thesis describes in detail how we implemented a system targeted at performing this kind of reasoning to planning and executing queries. In chapter 2 we provide a brief overview of the state of the art and common practices that pretain to query processing; in chapter 3 we describe how queries are processed and stored at a logical level; in chapter 4 we describe how FluiDB performs planning and garbage collection to come up with a concrete physical plan for each query; in chapter 6 we go over the algorithms and particular techniques that FluiDB employs to transpile the physical plan into C++ code; finally at chapter 7 we describe some exprimental evaluation of FluiDB on the benchmark SSB-TPC-H. We present a conclusion and some future directions in the final chapter 8

CHAPTER 2

Background

Something about the historical context of stuff

(Author)

Chapter summary

- Relational databases are question answering systems that deal with information organized in tables or relations.
- Query optimization and planning revolves around finding an efficient alorithm for answering a query.
- In-mamory databases keep all the data in main memory so they are closer to the processer
- many in-memory databases employ some variant of code generation to execute the query plans.
- Intermediate result recycling is the practice of reusing computation between queries.

FluiDB is a system that focuses on relational query, optimization and planning. This chapter aims to give the reader some idea about where FluiDB fits in the design space and the historical context in which it was developed. First, we outline a very high level overview of the query language and operators involved in database management, as well as the the overall architecture of such systems. Following that we will focus on the query planning subsystems of relational query databases and some traditional approaches to query evaluation. Afterwards, we will focus particularly on in-memory relational databases and the trade-offs that govern their design. Finally, we look into systems that utilize intermediate result recycling and how they solve the problem of automatically selecting and maintaining intermediate results for workloads.

2.1 Relational databases

Databases are more than just a method of accessing data. In their essence they are machines for question answering. The typical database works in a perpetual loop of reading queries and coming up with answers based on a set of data points. There are two important aspects that every database needs to define in order to delineate its operational semantics:

- The language in which the queries are expressed in
- The representation of data in terms of which the queries are expressed and the results are presented.

The oldest, most studied, and most common model is the relational model which defines queries in terms of *relational algebra* and organizes data in *tables* or relations.

A relation is typically an unordered set of tuples $(d_1, d_2, ..., d_k)$ where each element d_i represents an attribute. The core relational algebra is an extension of the algebra of sets (that defines operators of set union $\cup$, intersection $\cap$, product $\times$, and difference $-$) that includes the operators of joins $\bowtie$, projection π and selection σ. Upon this foundation relational databases typically define extra operators to increase the expressive power of the language like aggregations γ, semijoins $\ltimes$, sorting, limiting, etc.

A typical relational database processes operates in a cascading fashion. It initially receives a query in a textual form. While relational algebra is the language that underpins the operation of a relational database, it is rarely the language in which users interact with it. Instead, most relational databases expect queries written in a variant of SQL, a query language that is parsed into a tree of relational algebra operators.

This tree is processed by the query optimizer which rewrites it into a representation that is efficient to be executed by taking advantage of the mathematical properties of RA and often gathering statistics about the underlying data itself. This process leads to the formation of a relational algebra expression called the *logical plan*. It is a high level description of a sequence of operations required to produce a result. A logical plan only includes the denotational semantics of these operations, making no assumptions about their implementation.

Each RA operator is typically implemented by several different algorithms, each being efficient and even possible in some situations but not in others. For example, a join can be implemented by nested loops or with merge join. While the former algorithm is general and can implement any join, it is very inefficient. On the other hand a merge join is much more efficient but can only implement joins of the form $\bowtie_{a=b}$ (equijoins). Furthermore, a logical plan typically contains only implied information about the scheduling of the operators, for example, the logical plan $(\pi A) \bowtie (\sigma B)$ implies that the join can not begin being evaluated before the projection and selection but the latter can be evaluated in any order, or even simultaneously via pipelining. All these details about the execution of the query are resolved by the *physical planner*. The physical planner accepts a logical plan and emits an unambiguous algorithm that will produce the result of the query.

The final step is actually executing the physical plan which is handled by the *query execution engine*. The execution engine simply evaluates the physical plan on top of the data, making use of the correct algorithms, auxiliary structures like indexes managing memory, handling page-level caching, etc.

RDBMS diagram

2.2 Query optimization and planning

Query optimization and planning generally refers to the processes that take place between reading a query and executing a physical plan. The concerns of query optimization are the correctness of the final result and the efficiency of the plan generated, usually in terms of time, but also in terms of space.

Finding an optimal plan is in the general case NP-complete [12], but query optimizers can do a good job at finding good plans using heuristics. The selection and organization of these heuristics as well as query cost estimation are the main problems that make a query optimizer nontrivial. In particular one could separate the the tasks of a query optimizer into three broad categories:

- Query rewriting
- Query plan enumeration
- Size and cost estimation (cost model)

At a very high level, the architecture of a query optimizer is demonstrated in figure 2.1 and is broadly comprised of the following components:

- The *parser* which receives the query in textual form and produces a logical plan in the form of an abstract syntax tree.
- The *deterministic optimizer* or *logical planner* that rewrites the logical plan applying optimizations that are based solely on the general properties of the relational algebra.
- The *physical planner* that transforms a logical plan into a physical plan that can be unambiguously executed by the execution engine to produce a result. The physical planner typically has at least some information about the underlying data that the plan will operate on like estimations about the statistics of the values or the cardinalities of the relations.
- The information about the data being manipulated by the plan is inferred by the *cost estimator*. It uses a cost model to predict the cost of plans and the cardinality of relations by taking into account the provenance of relations as well as physical properties of the data like the presence of indexes, the data layout, etc.

These subsystems are presented here as separate for simplicity and because in many database systems they are clearly delineated, but it is also common that they blur into each other. For example, in some RDBMSes the physical and logical planner are merged into one [7, 37, 45]. In fact, it is increasingly common for systems to intersperse query planning with query execution, adapting the optimization strategy [4] to concrete information about the intermediate results evaluated rather than purely relying on estimations and predictions [21, 28, 51, 57]. The degree to which these subsystems are separate is a major concern in the design space of query optimizers.

Another important concern to be considered, and which is indeed important in the design of FluiDB, is the number of queries considered at a time during optimization and the way in which they are considered. The optimizer usually considers one query at a time and maintains little or no state between executions. Although it has found little adoption in main stream databases, multiquery optimization has been researched expensively [14, 41, 54, 68]. What is more common in recent years is recycling intermediate results. RDBMSes that incorporate this technique materialize and cache intermediate relations, reusing them when they appear as sub-queries in later queries [29, 40, 43].

The final query processing concern that is relevant to the design of FluiDB regards the traversal and pruning of the search space. As mentioned, query optimization is generally NP-complete, so the viable options DBMS designers are left with are randomized algorithms, ML approaches, and heuristics. Virtually all systems implement heuristics entirely or to some degree, while more and more also incorporate randomized algorithms [33] and machine learning [60, 67].

2.2.1 Logical and physical query optimization

The logical planner accepts a syntax tree in the form of relational algebraic expressions, where the operators only contain information about the semantic meaning of operations and none relating to the algorithms that will eventually be executed on the input data. It is for all intents and purposes a rewrite engine for relational algebra. Typical transformations performed by the logical query optimizer are

- Predicate normalization to conjunctive normal form, e.g. $(a_0 = 1 \land a_1 = b_1) \lor b_2 = 3 \hookrightarrow (a_0 = 1 \lor b_2 = 3) \land (a_1 = b_1 \lor b_2 = 3)$
- Predicate push-down e.g. $A \bowtie_{a_0=1 \land p} B \hookrightarrow (\sigma_{a_0=1} A) \bowtie_p B$.
- Cartesian product to joins $\sigma_p(A \times B) \hookrightarrow A \bowtie_p B$
- Searching the join ordering space.

The physical query planner, on the other hand, will specialize the logical operators deciding on the particular algorithm that should be used. A physical query planner therefore requires low level information about the query that relates to the indexes available, possible ordering of the data, materialized views, etc.

Either of the planners needs to enumerate the plans under consideration while traversing the search space. There are two major approaches to this:

- The *top-down* approach, where the planner establishes the top level operator and branches searching the children, backtracking as necessary. This was the approach of the descendants of the *volcano optimizer* [6] where they implement a "search engine" that uses a branch and bound approach to optimization with caching.

- The *bottom-up* approach where the planner builds and connects fragments of a plan accumulates those fragments into a larger plan in a traditional dynamic programming approach. [34, 62]

An important optimizer, on which most modern optimizers are still being based, is the *cascades optimizer* [7]. In short, cascades keeps track of *groups* of equivalent expressions of queries and uses those as the fundamental atom that it manipulates. For example instead of keeping track of $A \bowtie (B \bowtie C)$ and $(A \bowtie B) \bowtie C$ separately they would be part of the same query group. It then uses a global hash table (the "memo" structure) to match the best plan that corresponds to each group.

Many database systems use optimizers similar to cascades (like the Microsoft SQL server, Postgres, and MemSQL[50], and Greenplum – now Orca [46] to name a few). Notable among them is Apache Calcite [56], a framework for implementing query planning used by a number of commercial and research databases [63].

2.2.2 Cost estimation

Probably the hardest aspect of the planner design is the cost estimation algorithm. These are required in order to select plans and to navigate the search space. A good cost model can help basic planners make decent plans and sophisticated planners make horrible plans [49]. As cost estimation, we refer to a number of different related procedures that are broadly the estimation of the cost of an arbitrary plan and include the prediction of the cardinality of not yet materialized relations.

It seems that the most important challenges involved in the design of a cost model relate to the fact that we are fundamentally operating with scarce and highly uncertain information. Especially relating to cardinality estimation, the most important of the tasks involved, uncertainty and bad predictions propagate and make make it extremely hard for the planners to make correct decisions. Consider, for example a join. Some joins are similar to Cartesian products, producing large output tables, and some joins are more similar to lookups producing only a few rows. A cost model that confuses the kind of join will make very bad predictions w.r.t. the cost of any relational algebra expression that uses the said join.

Cost estimation is beyond the current scope of FluiDB and we only implement the most naive cardinality estimation, but it is worth mentioning how more mature systems approach the issue. Cardinality estimation most commonly takes the form of selectivity estimation, i.e. what percentage of tuples from the input make it to the output of a selection or a join. This is sensitive to the statistical properties of the values and the selection/join predicates.

The most common approach to this issue is to maintain pre-computed statistics related to the primary tables. This yields better plans than keeping no statistics at all, but they are very hard to maintain and their effectiveness becomes very limited for complex queries. Another approach employed is to delay the decisions of the optimizer and essentially merge the planning and plan execution process. This does not completely eliminate the problem, but it allows the system to have more up to date and precise data that would allow it to correct course early in the event of exceptionally bad estimations. One family of techniques that is becoming popular and was initially used in IBM DB2 [16], is caching the statistics of previously computed relations and using that data to make better predictions in the future. More recently, this takes the form of employing machine learning to learn from past cardinalities [61].

To give the reader a more well-rounded intuition of the state of cost estimation besides cardinality estimation, it is worth mentioning that, to estimate the cost of an operation, Postgres uses magic configurable variables to weigh IO with CPU evaluation time, while DB2 runs micro-benchmarks on the production system to make these estimations.

2.3 In-memory relational databases

In-memory databases are databases where all data lives in main memory. The design of in-memory databases is different from the design of a disk-backed database in a number of respects. To name a few:

- Page buffers have little use and caching of data in general has very different goals. While in disk-backed databases caches are mainly used to avoid disk IO, in in-memory databases they focus on reuse of computation.

- Concurrency control is much simpler as storage synchronization concerns are almost entirely eliminated.

- In disk-backed databases, only a small percentage of time is spend on actual computation [58]. Much of the non-compute latency is directly linked to the persistent storage.

On the other hand, there are concerns that are specific to the lack of backing database storage. To name a few:

- Should the system rely on record IDs like in a persistent database or can it use direct pointers to records?
- Error prone software can bring the data to an irrecoverable state.
- Query execution algorithms have fundamentally different characteristics. While in persistent databases IO dominates the runtime the bottlenecks for an in-memory database are much more complex and they can include things like locking, cache misses, predicate evaluation, and data movements.
- As main memory is a much less abundant resource than persistent storage, in-memory databases are often distributed, making network a major concern.

One increasingly common technique to address many of these issues, which is also used by FluiDB, is **code generation**. Since workloads for in-memory databases are typically CPU bound, there are major gains in performance to be had by specializing the code being executed. The typical generic code being run makes heavy use of virtual function calls and conditionals inside tight loops, which kills performance on virtually all modern architectures. The value proposition of code generation is to inline or hard-code the virtual functions and erase the conditionals at runtime to reduce the number of operations and make better use of hardware optimization.

We identify 4 different approaches in the literature to solving this problem:

2.3.1 Transpilation

Transpilation of a physical plan to a systems language like C or C++ which is then fed to an off-the-shelf compiler [30]. This approach is expensive but it generates highly efficient code more easily debuggable execution plans. The most notable complete database systems that used this technique were Microsoft Hekaton that generates C code from

SQL queries and older versions of MemSQL. For FluiDB, we reuse a lot of techniques introduced in HIQUE [30] to translate physical plans to template-heavy C++ code, making the assumption that query compilation will be much faster than the query runtime.

2.3.2 Third party JIT compilers

JIT compilation has received a lot of attention in the compiler community, especially in the context of the JVM. A number of database systems have taken advantage of this to speed up query execution. Most notable of these are SPARK [47], which generates Scala AST which is then converted to JVM byte code, and Neo4j that directly generates bytecode out of the queries.

Systems like Peloton [53] and recently Postgres [55] compile query plans to LLVM IR code that is then fed to the LLVM compiler to generate high performance machine code. Notable among the systems that use this approach is HyPer [77] which addresses the problem of query compilation overhead with an adaptive execution approach: they built an IR interpreter that starts running the query while the compiler does proper optimization and compilation of the LLVM IR program being interpreted. Cheap queries are thus completed reasonably fast, while in the case of complex queries the interpreted program is seamlessly replaced by the compiled one once the LLVM compiler finishes generating machine code.

2.3.3 Direct machine code generation

Some databases do not reuse any compiler or JITing VM, but rather directly generate highly specific machine code out of the physical plan. The first system to attempt that, like most techniques used today, was System-R, which originally compiled SQL statements directly to machine code by stitching together code fragments from a "fragment library" [2] but it was quickly deprecated due to the large engineering effort required. Oracle [ref] also includes similar to some of their databases and MemSQL express their plans in a custom language called MPL (MemSQL Plan Language) for which they have a custom compiler that translates it directly to machine code.

2.3.4 Custom execution engines

The final category is per database code generation rather than per-query code generation. Volcano/EXODUS [6] and more recently SageDB [75] generate an optimizer and execution engine that is specific to the database schema but not to the queries.

2.4 Intermediate result recycling

A materialized view is a relation defined by a query that is persistently stored. A view that is not stored is said to be virtual. **View selection** is the process of selecting an appropriate set of materialized views to improve the performance of a workload [36]. Automated materialized view selection or intermediate view recycling has been on the radar of the database research community for a while now. A few approaches to this problem have to do with AND/OR directed acyclic graphs [9], modeling the problem as a state optimization [11], and lattices to represent data cube operations (i.e. multiple aggregations over the same relation) [72].

A related problem is multiquery optimization [11] that attempts to plan multiple queries simultaneously. An efficient solution using AND/OR DAGs was proposed by Roy in [14] where they insert queries and their sub-queries in a graph and attempt to evaluate a plan by traversing that graph from multiple sources in a fashion similar to volcano optimization. Building on that Kathuria et. al. present an approximation algorithm that runs in time quadratic to the number of common subexpressions and provides theoretical guarantees on the quality of the solution obtained [52].

Researchers have further looked at opportunistically reusing intermediate results that would be materialized [29, 40]. It has been researched in non-strictly database contexts like MapReduce [35], and there have been attempts to unify the planner with the materialized view selection engine [44]. Notable in this field is Nectar [27] which is also not an RDBMS. It automatically compresses rarely used data into programs that generate that data. Nectar focuses on sharing computation and data as much as possible.

Work such as MRShare [31] tries to bridge the gap between intermediate result recycling and MQO by automatically grouping queries in a workload in such a way that computation can be maximally shared.

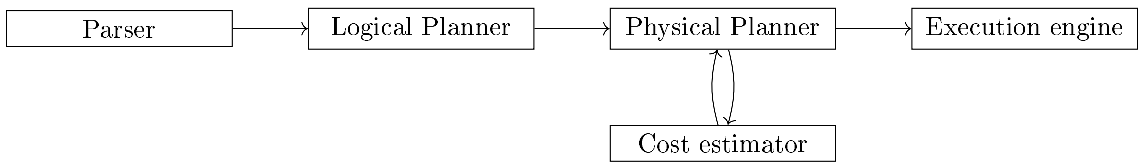


Figure 2.1: Common architecture of a query optimizer.

CHAPTER 3

Logical planning

Something about accumulating relationships.

(Zizek)

Chapter summary

- FluiDB expresses queries in relational algebra and defines reverse operations for most operations.
- Query plan trees are accumulated in a bipartite DAG (QDAG) where the types of nodes are tables and bidirectional operations.
- Queries are normalized to hashable objects (QNF) that abstract a wide range of rewrites. This allows efficient merging query plan trees into the QDAG.
- Sub-graphs of the QDAG (clusters) represent higher level operations and are used as *propagators* to infer information of the n-nodes like cardinality, column type, and uniqueness of subtuples.

FluiDB is focused on query optimization and planning when the queries in a workload being served can have common subqueries. To exploit these similarities between queries, FLuiDB transforms them to bipartite graphs of subqueries and bidirectional operators, which she incrementally merges into a global query DAG, which it then uses to enumerate the possible plans. In this chapter, we will dive into how FluiDB processes and merges these graphs.

We will start by looking into what kinds of queries FluiDB can understand and how she processes them. Then we will look at how she relates the queries to each other to construct a tightly interrelated story of the workflow by looking at the basic query graph, the heart of FluiDB.

For manipulating a graph of queries, it is important that FluiDB can distinguish between semantically different queries and when they are the same. We will look at how FluiDB normalizes queries to efficiently determine this equivalence.

Then we will see how FluiDB organizes the nodes in the query graphs into clusters representing single operations are formed within the query graph and how those are used to maintain consistency among information about intermediate results.

Finally, we conclude by describing how the query processing infrastructure, which sits at the core of the RDBMS, interfaces with the rest of the components: the planner, Antisthenis and the code generator, described in later chapters.

3.1 Relational Algebra

Much like most other RDBMSs, FluiDB deals with queries expressed in terms of a *relational algebra*. Here we define the FluiDB relational algebra semantics, that is, the traditional relational algebra extended by unary operators for ordering $s_e(X)$, limiting $l_i(X)$ and aggregating $\gamma_e(X)$, as listed in table 3.1.

Table 3.1: Correspondence between relational algebra expressions and SQL.

Description	RA	SQL
Aggregation	$\gamma_e(X)$	`select * from X group by e`
Ordering	$s_e(X)$	`select * from X order by e`
Limit	$l_i(X)$	`select * from X limit i`
Join	$X \bowtie_p Y$	`select * from X, Y where p`
Anti-semijoin	$X \not\ltimes_p Y$	
Union	$X \cup Y$	
Selection	$\sigma_p(X)$	`select * from X where p`
Projection	$\pi_r(X)$	`select r from X`

Before going any further, it is important to clarify some assumptions made to accommodate the design of FluiDB. Particularly that $\forall A, p.\sigma_p A \cup \sigma_{\neg p} A \equiv A$. This means that there no `NULL` values are allowed since a predicate over a NULL value always evaluates to `False`. This allows the definition of a bidirectional variant of the select (σ) operator.

Further, we assume set semantics for all relations, meaning that a valid relation needs to have at least one unique sub-tuple. The user is not burdened with the responsibility of enforcing this, it is rather enforced by a pre-processing stage on projections and aggregations, exposing extra columns than the ones specified by the user such that the resulting relation has at least one sub-tuple that is unique for each row (see section 3.3 on query pre-processing).

The semantics of the sorting operation are preserved by making a small concession to the set semantics: The *only* time when records of a relation are assumed to be ordered is for expressions of the form $s_e(X)$ and only for the duration between it running and the next operation. In practice, this means that the FluiDB planner treats s like an identity operation.

Besides these caveats all operators known to FluiDB behave as expected. The first of the extensions adopted is the antijoin operator $\not\ltimes_p$. It is equivalent to $A \not\ltimes_p B \equiv A \backslash \pi_A(A \bowtie_p B)$ and it is primarily useful for creating a bidirectional join by capturing the rows left over from a join (see subsection 3.2 on the QDAG). The important property we will be interested in is:

$$A \not\ltimes_p B \cup \bar{\pi}_{cols(A)}(A \bowtie_p B) \equiv A$$

$\bar{\pi}_{cols(A)}$ means "group by the unique columns of A" As syntactic sugar we also define:

$$A \not\rtimes_p B := A \not\ltimes_{\neg p} B$$

3.1.1 Expressions

Relational algebra expressions contain expressions that operate at the field level. These appear in projection column definitions, selection predicates, aggregation column definitions, etc. These expressions are organized in up to 4 nested layers based on the kinds of operators:

- The top layer is purely Boolean operations `Prop` that can be rewritten in Cartesian normal form.
- The intermediate layer `Rel` describes relations between values ($\equiv$, `like`, $<$, $\leq$, etc). This layer is not nested, i.e. we do not allow relations between relations.
- The lowest layer `Expr` is expressions of any kind that may return any value. The above layers can be expressed in terms of this layer to support the ternary operator `if <expr> then <expr> else <expr>`.
- The symbol layer describes the literal and symbolic atoms involved in the query. Its type is parametric in most parts of FluiDB to allow us to tag the symbols in the expression without breaking compatibility.

Predicates of selections and joins (joins are always θ joins at the type level) have type `Prop (Rel (Expr e))` for some type `e` and projections have type `QProj [(e,Expr e)]`. A special layer is provided for aggregations `Aggr` that makes it possible to disallow nested aggregation functions at the type level making the aggregation operator `QGroup [(e,Expr (Aggr (Expr e)))] [Expr e]`, where the first argument of the constructor is the aggregation functions and their names, and the sectond argument is the expressions on which the grouping should happen.

The detailed definitions of the algebraic expressions are presented in listing 3.1.

```
-- | Algebraic type of a proposition
data Prop a = P2 BPOp (Prop a) (Prop a)
  | P1 UPOp (Prop a)
```

```haskell
  | P0 a
data UPOp = PNot
data BPOp = PAnd | POr

-- | Algebraic type of an expression
data Expr s = E2 BEOp (Expr s) (Expr s)
  | E1 UEOp (Expr s)
  | E0 s
data UEOp =
  EFun ElemFunction
  | ENot
  | EAbs
  | ESig
  | ENeg
data ElemFunction
  = ExtractDay -- erased after parsing
  | ExtractMonth -- erased after parsing
  | ExtractYear -- erased after parsing
  | Prefix Int
  | Suffix Int
  | SubSeq Int Int
  | AssertLength Int
data BEOp =
  -- Boolean (truthy)
  EEq | ELike | EAnd | EOr | ENEq
  -- Numeric
  | EAdd | ESub | EMul | EDiv

-- | Algebraic type of a relation.
data Rel a = R2 BROp a a
data BROp
  = REq
  | RGt
  | RLt
  | RGe
  | RLe
  | RLike
  | RSubstring

-- | Algebraic expression of an aggregation function
data AggrFunction = AggrSum
  | AggrCount
  | AggrAvg
  | AggrMin
  | AggrMax
  | AggrFirst
```

Listing 3.1: Types of algebraic expressions that compose the non-RA parts of the queries.

3.2 QDAG: The unified query graph

The operation of FluiDB revolves around an AND-OR graph of relations and algebraic operations, not unlike the ones commonly used in multiquery optimization literature. This graph encodes all plans that the query optimizer and planner will enumerate by means of traversal. Said graph is a bipartite graph where t-nodes (AND nodes in other literature) represent relational operations and n-nodes (OR nodes elsewhere) represent relations that will act as intermediate results. A set of extensions is introduced to the traditional AND-OR DAG to facilitate the operation of FluiDB. In the interest of motivating those extensions, we will take a very high level view of the operations on the QDAG before describing them in detail.

There are multiple ways of composing relational operations to materialize a certain relation. Each one is a relational algebraic representation and we commonly refer to them as logical plans. Each one represents a tree similar to an abstract syntax tree. These trees are made of two kinds of nodes: operation nodes – t-nodes, and relation nodes – n-nodes. For example, the query $A \bowtie B \bowtie C$ may be represented as $(A \bowtie B) \bowtie C$ or $A \bowtie (B \bowtie C)$ due to associativity of $\bowtie$. Those two expressions represent two tree forms where the join operators $\bowtie$ are represented as t-nodes and each of A, B, C, $A \bowtie B$,

$B \bowtie C$ and $A \bowtie B \bowtie C$ are represented as n-nodes.

Each n-node can be materialized by *triggering* one of the input t-nodes, by running a connected operation to which said n-node is on the output. It is clear then where the name OR-node found in other literature originates: any of the input nodes may participate to a plan involving the OR-node (n-node for us). Conversely, each t-node can be triggered $\iff$ all of it's input n-nodes are materialized. Because *all* input nodes must be materialized, t-nodes are elsewhere referred to as AND-nodes. We depart from the standard nomenclature to avoid confusion in the presence of the extensions introduced by FluiDB. The planning of a query amounts to searching for a sequence of t-nodes to be triggered that would result in the corresponding node being materialized.

The QDAG of a query is created by unifying all logical plans that would derive it by means of merging n-nodes that are equivalent. For example the mentioned trees $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ shared the nodes $A \bowtie B \bowtie C$, A, B and C. A global QDAG is maintained by unifying the QDAG of each query encountered into the global QDAG, again, by merging equivalent n-nodes.

From a QDAG we can derive logical plans for queries. The planner operates in terms of the global QDAG. It traverses the QDAG searching for a plan (i.e., a sequence of t-node triggers) that will cheaply materialize the requested node(s). This way, historical nodes can be naturally considered. The problem of reusing materialized intermediate results is thereby conceptualized as a graph traversal problem.

During this planning process, if storage space runs out, the garbage collector (GC) is triggered. Garbage collection in this context is the process of deleting materialized relations that are not used for the query at hand. We shall say that an n-node is *deleted* when the garbage collector appends to the plan a delete operation that removes the corresponding materialized relation from storage.

This technique is heavily inspired by multiquery optimization (MQO) approaches like [15] where they use the AND-OR DAG to solve multiple queries simultaneously, optimizing computation reuse.

QDAGs are different from AND-OR DAGs in a subtle but important way: each t-node may have multiple outputs and can be partially bidirectional. In other words, a t-node, as well as multiple inputs, may have multiple outputs, any number of which may be materialized by it being triggered. A *reverse trigger* of said t-node would materialize inputs provided that all outputs are materialized. In other words, where AND-nodes are similar to functions that when evaluated produce an output given some input, t-nodes correspond to bidirectional relations between sets of input and output relations.

This not only creates opportunities in terms of what plans are possible, but also opens the gate to a whole new regime for how garbage collection is involved in the process of query planning. Any n-node may potentially be garbage collected, even primary tables, as long as the garbage collector can prove that it is reconstructable by triggering or reverse triggering t-nodes. The storage budget is thereby taken into account not only when selecting or rejecting materialized views, but also while planning the query solution itself.

To make this process clearer, as an example, consider a DAG that contains just a selection:

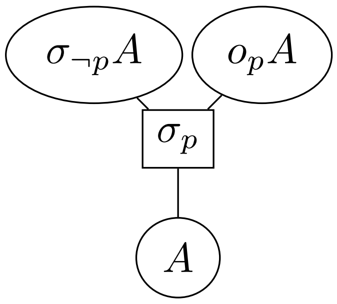


Figure 3.1: A selection t-node may generate both output nodes, rendering it a partition operation.

In the figures throughout this work we will denote n-nodes as circular and t-nodes as squares or rectangles. When materializing $\sigma_p(A)$ the planner has the option of also materializing $\sigma_{\neg p}(A)$. If deemed beneficial A can be safely deleted as the t-node can be reverse triggered to produce it. A reverse trigger amounts to the union $\sigma_p(A) \cup \sigma_{\neg p}(A)$. It is generally assumed that any row not selected by p is necessarily selected by $\neg p$. To remind the reader of our RA assumptions, this means that we do not support missing or **NULL** values as they break Boolean double negation. Furthermore, relations are taken to be bags of tuples, i.e., they are unordered.

A slightly more complex example might be

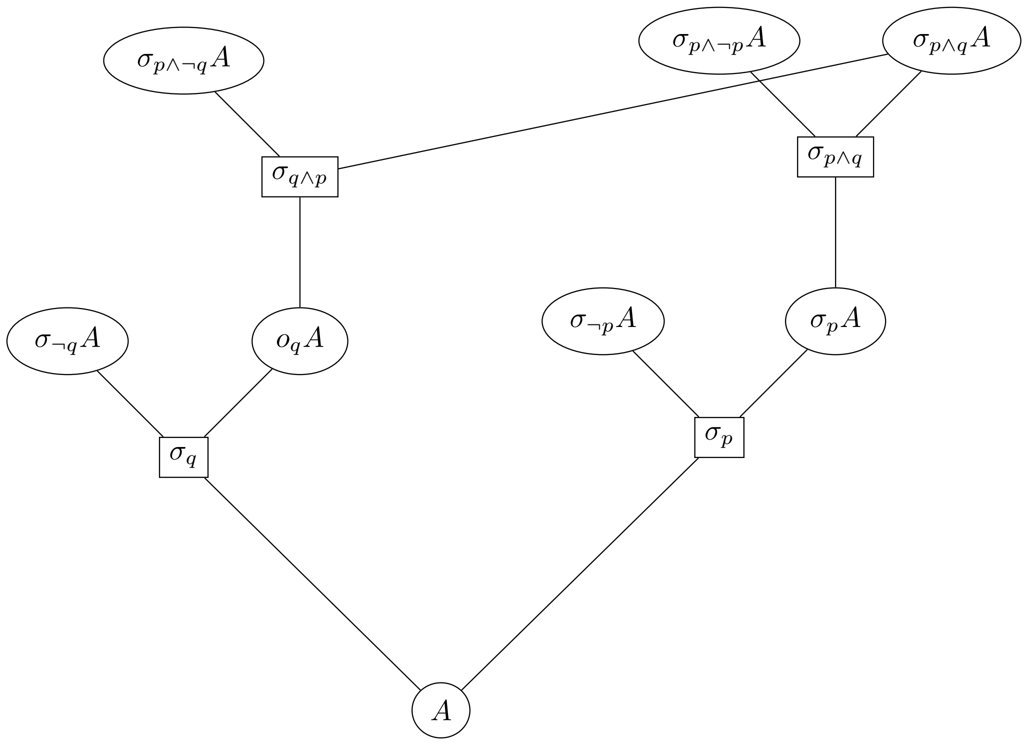


Figure 3.2: Multiple captions can be composed to create larger networks.

Here if $sigma_{\neg p}A, \sigma_q\sigma_p A, \sigma_{p\wedge\neg q}A$ are all materialized, then A is materializable as their union. This way $\sigma_p\sigma_q A$ is readily available with no space overhead. Instead, the price paid is that in the event that A is required, possibly in a different context, constructing it requires some overhead. Furthermore, there are two options for materializing $\sigma_p A$: either by selecting over A, or via the union $\sigma_{q\wedge p}A \cup \sigma_{\neg q\wedge p}A$. Table **??** demonstrates the correspondence between relational algebra operators and the equivalence that enables their reverse

Table 3.2: Each operation that may end up in the final result and the equivalence that ensures reversibility. $\perp$ values indicate that the operator is not generally reversible.

Operator	Equivalence
Aggregation ($\gamma_e(X)$)	$\perp$
Sorting $s_e(X)$	$\perp$
Limit ($l_i(X)$)	$l_i A \cup d_i A \equiv A$
Join ($X \bowtie_p Y$) / Anti-semijoin ($X \not\ltimes_p Y$)	$\bar{\pi}_{cols(A)}(A \bowtie B) \cup (A \not\ltimes B) \equiv A$
Selection ($\sigma_p(X)$)	$\sigma_p A \cup \sigma_{\neg p} A \equiv A$
Projection ($\pi_r(X)$)	$\pi_r A \bowtie \pi_{\bar{r}} A \equiv A$

3.2.1 Clusters and irreversible edges

A concept closer to the functionality of t-nodes and n-nodes than AND and OR nodes, and in fact our very inspiration for this approach, is *propagators* [24]. In short, propagators are hyperedges in a hypergraph, the nodes of which contain partial information about a value. Information about a value can be accumulated from its neighbors. Each node can obtain at least as much information about its value as the most precise neighbor. The amusing yet illustrative example provided in the paper is the one of measuring the height of a building by means of a barometer. One approach is timing the barometer's free fall from the top of the building, another is by measuring the length of the shadow of the barometer and comparing with the length of the shadow of the building at the same time of day. One may even use the barometer in its intended function and derive the height of by comparing the difference in air pressure at the roof and at the ground floor. Each of the mentioned methods will give very noisy or partial information about the magnitude in question. In the propagator paradigm, each of these measurements might be represented as a node in a hypernet. A propagator edge connecting these nodes would combine information from all measurements and improve each of the nodes' accuracy in their perception of the underlying value. A t-node is similar to a propagator in that it does not have a strict direction of information flow and can generate information on either side given information on the other.

Generally, there is no one-to-one correspondence between t-nodes and relational operators. Usually that is the case but there is an exception: the join operator. As we made clear, a t-node can generate any number of its inputs by reverse triggering it as long as all of the outputs are materialized. Then one approach to implementing a join operator might be:

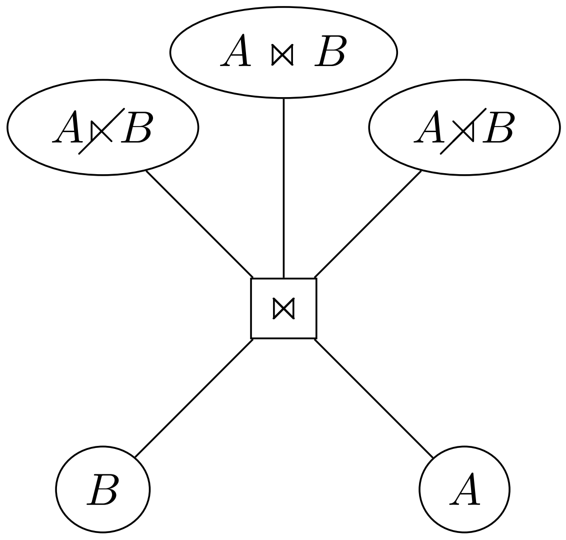


Figure 3.3: Representing a reversible join as a single t-node.

The $\not\ltimes$ operator is the antijoin operator, $A \not\ltimes B$ includes the tuples in A that to not participate in $A \bowtie B$. It is the complement of the semijoin $\ltimes$. Therefore

$$\bar{\pi}_{cols(A)}(A \bowtie B) \cup (A \not\rtimes B) \equiv A$$

We remind the reader that $\bar{\pi}_{cols(A)}$ means group by the unique columns of A. Therefore, by having materialized $\{A \not\rtimes B, A \bowtie B, A \not\ltimes B\}$ we can create A, B and vise versa. We could, however be slightly more flexible than that, as we might want to materialize $A \rtimes B$ but not $A \not\ltimes B$ to the end of ensuring the materializability of only A and not B. We therefore represent joins not as a single t-node but rather as a *cluster* of t-nodes.

However, in reality we can afford a bit more flexibility than choosing between the input and the output sets: We swap $B \not\rtimes A$ with B in these sets or $A \not\rtimes B$ with A, thereby having materialized the tables $\{A, A \bowtie B, B \not\rtimes A\}$. In this case neither the entire input nor the entire output are materialized and yet all tables involved are materializable. To overcome the limitation that this poses, we make the graph a bit more complex and a bit more flexible by breaking the join into two stages (see figure 3.4):

- First, we separate the join components (A and B) into the part that will be used in the join ($A \not\ltimes B$ and $B \not\ltimes A$ respectively), and the part that will not ($A \not\rtimes B$ and $B \not\rtimes A$ respectively).

- Then we join only the useful parts $(A \not\ltimes B) \bowtie (B \not\ltimes A) \equiv A \bowtie B$.

The left antijoin node partitions A into $A \ltimes B$ and $A \not\ltimes B$, which is left semijoin in our null-less context. Similarly for the right antijoin node which partitions B to $A \rtimes B$ and $B \not\rtimes A$. Joining the semijoin relations $A \ltimes B$ and $B \ltimes A$ is equivalent to joining A and B.

In figure 3.4 we used an antijoin t-node. From the semantics of the antijoin $A \ltimes B$ it is clear that no tuples from B are included in either $A \not\ltimes B$ or $A \not\rtimes B$ as B acts as a kind of predicate for selecting over A. This means that even though B is an input, there is no natural way of creating B from the t-node's output. For these cases we introduce *irreversible edges*. Remember that edges are always connections between n-nodes and t-nodes. An irreversible input edge (a connection from an n-node to a t-node) can only be used as an input and its n-node can not be materialized by reverse triggering the t-node.

Irreversible edges appear in the inputs of anti-join t-nodes but also on the input of operations that cannot be information preserving, notably aggregations. The output of an aggregation can not in general be supplemented with extra information to reconstruct the input, unless the entire input table is replicated. We therefore give up the reversibility of such operations completely and mark the input edge as irreversible. This is generally not a big problem as on the one hand the results of aggregations tend to be small compared to the input relations, and in the case of anti-join t-nodes, which are only found in join clusters, for every anti-join node $A \not\ltimes B$ that is unable to generate B by reverse triggering, there is a $A \not\rtimes B$ that will happily generate B albeit refusing to generate A. This is demonstrated in figure 3.5.

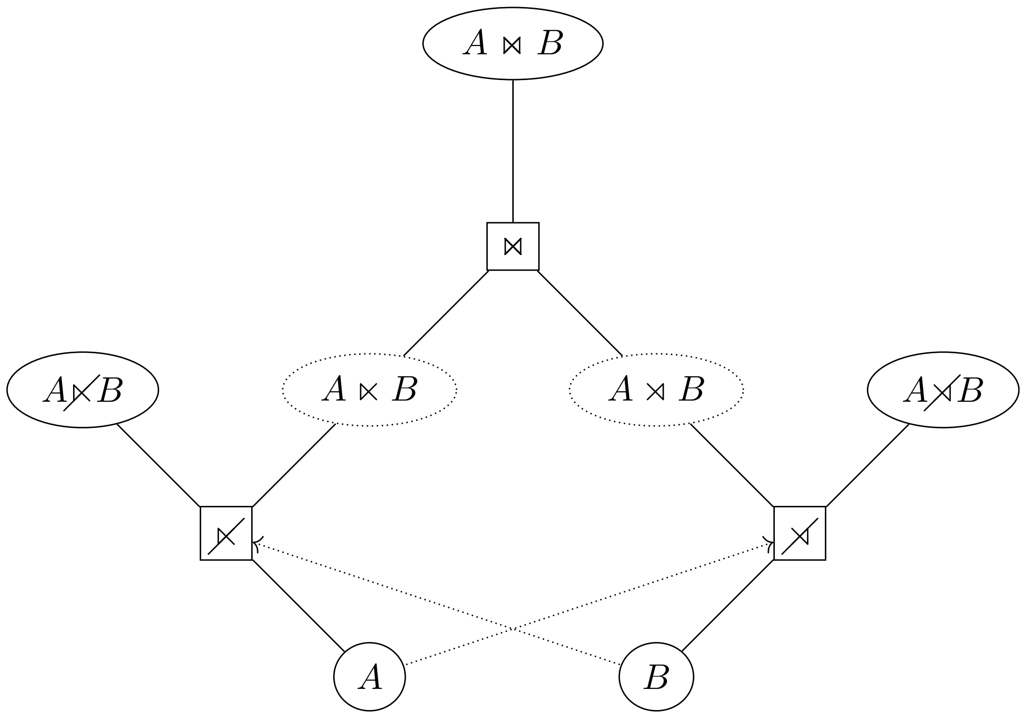


Figure 3.5: The QDAG corresponding to a join is broken down in multiple t-nodes to facilitate greater flexibility.

3.3 Deterministic query processing

In this section, we will look at the initial steps of what happens when FluiDB receives a query. Initially, the query is accepted in textual form as SQL. Then it is sanitized to adhere to the properties we defined for relational algebra and some obvious optimizations are applied to remove artifacts of the parsing process. Finally, the query is translated to a forest of possible join orderings. These are rewrite-based optimizations that are deterministic and do not take into account the rest of the workload.

3.3.1 Query parsing

We use a custom library for parsing SQL queries into relational algebra based on parser combinator monads [76]. Our parser handles both the parsing and decorrelation of the queries. In brief, parser combinators is a functional interface to combining parsers to make larger, more complex parsers. A parser `p` has the form `p a` which means that if the parser is used to parse the matching text, it will return a result of type `a`. For this framework to work, two combinators are required:

- sequence (monadic bind) `(>>=) : p a → (a → p b) → p b` which sequentially tries two combinators and the latter one may use the result of the former.
- the 'alternative' parser combinator `<|> : p a → p a → p a` which tries one parser and if that fails tries another.
- `MonadGen e p` can generate a value of type `p e`. Useful for creating names for unnamed expressions in projections, groups.

Simple parsers like the `parseInt` parser combinators are then combined to create more complex parsers (listing **??**). This approach is fundamentally not dissimilar to writing yacc files but it is much more flexible as parser combinators effectively function as a domain-specific language that inherits all the power of the host language, in our case Haskell.

```
parseInt :: MonadParse p ⇒ p Integer
parseInt = do
  sign ← fmap (maybe 1 (const $ negate 1)) $ parseMaybe $ char '-'
  void $ parseMaybe $ char '-'
  (* sign) . read <$> readWhile isDigit

parseSelectProj
  :: (MonadGen e p,MonadParse p)
  ⇒ p e → p (Query e s → Query e s)
parseSelectProj eM = do
  word "select"
  fmap (Q1 . QProj) $ sep1 (word ",") $ do
  ex ← parseExpr (E0 <$> eM)
  parseMaybe (word "as" >> eM) >>= \case
    Just e → return (e,ex)
    Nothing → do
  e ← maybe pop (return . mkSymbol) $ asSymbol ex
  return (e,ex)
```

Listing 3.2: The selection parser is much more complex and handles many different cases, but it is built up from simple fundamental blocks. This parser returns a query modifier that is meant to be applied to a very simple product query generated by the `from` clause.

3.3.2 Query prepossessing

Immediately after parsing, the queries have several potential inconsistencies with the notion of relational algebra that we described in section 3.1. For that reason, we implement an intermediate stage where we rewrite the query to comply with various properties of the RA, and which remove redundancies. In brief, those are the following:

- Projections and selections are squashed. For example $\sigma_p \sigma_q$ becomes $\sigma_{p \wedge q}$.
- Date intervals are translated to dates. This is because the physical planner does not handle dates specially so all dates and date intervals are converted to timestamps and second intervals. For example the expression `date '2020-12-01' - interval '90' day` is immediately turned into `date 2021-03-01`. Note that an expression like `date '2020-12-01' + shipping_time` is not supported as it is not date arithmetic between literals.
- Optimization of the `like` operator: `<VAR> like <STRING_LITERAL>` expressions where `<STRING_LITERAL>` begins or ends with `\%` are turned into `suffix_of` and `prefix_of` expressions. For example, the expression `a like "\%.txt"` would be turned into `".txt".suffix_of(a)`.
- Substring expressions are turned into 0-indexed ones from 1-indexed as is the case in SQL.
- It is asserted that each intermediate relation has uniquely named columns. For example, there are necessarily name conflicts in $A \bowtie A$.
- It is asserted that sorting appears higher in the query AST than a selection, a join or an aggregation. All three of these operators affect the ordering of their inputs (see chapter 4 on code generation), and sort is the only operation that asserts tuple ordering on its output. Therefore $s(A \bowtie B)$ is valid but $s(A) \bowtie B$ is not.

Once the preliminary sanitization and simple optimization steps are done we mark each instance of the primary table symbols with the columns and their types. This is simply information derived by the database schema. Then we do some processing and annotation on symbols referring to symbols:

- First *remap unique sub-tuples*. Each intermediate result, and each relation that we want to be able to reason about needs to have uniquely identifiable rows. For each relation in the QDAG we are keeping track of a set of columns, the unique sub-tuple, the combination of which is unique among the tuples of the same table. It is fairly easy to keep track of the unique subtuple for most operations, e.g., the unique sub-tuple of a Cartesian product or join is generally the concatenation of the unique sub-tuples of the input tables, the unique sub-tuple in the case a selection is the same as the unique sub-tuple of the input, in aggregations the unique sub-tuple is the same as the columns on which we aggregate, etc. The main problem is projections and, in a similar manner aggregations, where it is possible that the operation does not project on the entire unique sub-tuple. To mitigate this, we perform a pre-processing step to rewrite the projections to expose the entier unique sub-tuple. It is worth mentioning that there are cases where there are more than one combinations of columns that form a unique sub-tuple. We opt to keep track of all of them in order to have more flexibility in this pre-processing step: ideally we want to modify the projection as little as possible. In other words we want to project as few extra columns as possible to keep the table size as small as possible.

- Annotate each symbol referring to a column with the corresponding primary table if there is one (named columns of projections do not correspond to a primary table). This is useful both in determining possible joins (as we will see in section 3.3.3) and in determining the types of columns.

- Infer and annotate the type of each column reference. This inference is sometimes trivial, for example, in the case of references to columns of primary tables, or when annotating literal values, but when references to columns of projections it is more complex. For example, in the query $\sigma_{p(a)}\pi_{a\mapsto f(X)}$, we need to infer the type of a. FluiDB will apply some very simple heuristics to fake proper type inference. This is usually enough to come up with accurate types as SQL's type system is basic.

- Projection sharing is determined. As we will see in detail in section 6.4.3 where we will be discussing the implementation details of the projection algorithm and its reverse, it is important that at least one unique tuple of a projection is shared between a projection and its complement. At this stage, it is decided which unique subtuple is to be shared and it is recorded in the RA expression tree.

3.3.3 Possible joins

Join ordering is one of the most detrimental optimizations of a query. It is the selection of the associative order of joins. The relation $A \bowtie B \bowtie C$ can be exressed as $(A \bowtie B) \bowtie C$ or as $A \bowtie (B \bowtie C)$. Individual joins can have a wide range of behaviors. From being similar to Cartesian products, being prohibitively costly, and exploding the cardinality of the result; to being similar to lookups, very cheap, and with a very small result. Optimizers that process queries in isolation can aggressively push down selections and prune the search space of join combinations when they encounter expensive intermediate results. FluiDB, however, needs to keep track even of more expensive intermediate results as the high cost may be amortized in the long run.

For this reason, at this stage we enumerate all possible join permutations building a graph of all possible joins.

- All joins are turned into selection-products, so $A \bowtie B$ becomes $\sigma(A \times B)$
- selections are pulled up and merged, ($\sigma_p \sigma_q A$ becomes $\sigma_{p \wedge q}$)
- products are represented as sets of relational algebraic expressions.

Then all possible permutations of joins are created. For example $\sigma a = b \wedge b = c(A \times B \times C)$ becomes $\{(A \bowtie_{a=b} B) \bowtie_{b=c} C, A \bowtie_{a=b} (B \bowtie_{b=c} C)\}$.

Note that we do not include queries that are only different w.r.t. commutativity, i.e., we don't include both $A \bowtie B$ and $B \bowtie A$, this part of FluiDB depends on the consumer of the data structure to disambiguate between them. Another caveat is that since Cartesian products are very large and are almost never used in the actual plans, an option that disallows product intermediate results is enabled by default. Therefore, in our example we prune relations like $\sigma_{a=b}((A \times C) \bowtie_{b=c} B)$. Furthermore, allowing Cartesian products exponentially explodes the space of possible plans to a degree that makes the resulting data structure virtually unusable.

Once selections are merged, we find product trees, i.e., contiguous parts of the tree that only contain Cartesian products, and combine them into multi-sets of subtrees such that $\sigma(C \times (A \times B) \times D)$ becomes $\sigma(\{A, B, C, D\})$.

Then selection propositions are normalized to conjunctive normal form and into sets. For example $\sigma_{p_1 \vee (p_2 \wedge p_3} A$ becomes $\sigma_{(p_1 \vee p_2) \wedge (p_1 \vee p_3)} A$, and in fact the conjunctive terms are also represented as sets of subterms, so the final form would be better expressed as $\sigma_{\{p_1 \vee p_2, p_1 \vee p_3\}} A$. This leaves us with a tree of terms of the form $Q[\sigma_{\{p_0, p_1, \ldots, p_l\}}\{Q_0, Q_1, \ldots, Q_k\}]$ where each term p_i refers to columns in one or more Q_j relations and $Q[\cdot]$ is a query tree with $\cdot$ as a subtree. We will refer to such a selection-product pair as an SP-term.

To turn this representation into a forest, we will turn each SP-term into a set of regular join trees, the Cartesian product of which will yieled the possible joins. We summarize the process as follows:

- We iterate over all pairs of sub-multisets into which the multiset $\{Q_0, Q_1, \ldots, Q_k\}$ can be split. We follow each of the following steps for each of the resulting sub-multisets.

- We split the p terms in three parts depending on which sub-multiset of relations their symbols refer to. One set are terms where the columns mentioned refer *only* to the left sub-multiset (left p-terms), another set are terms where columns mentioned refer *only* to the right sub-multiset, and the rest (right p-terms) must necessarily have references to columns of *both* sub-multisets. We call the latter category *connector p-terms*.

- We turn each side of the split into an SP-term by associating it with the left and right p-term set. We repeat the process of generating join trees out of each SP-term created and connect them in an all-to-all fashion by creating theta joins using the conjunction of the connector p-terms. In case the the set of connector p-terms is empty, that would mean that the connection of the join trees must be done with a Cartesian product. As mentioned above, we want to avoid inserting Cartesian products in our graph so we reject those cases outright. Optionally, we may even want to reject all non-natural joins altogether, unless that would mean not coming up with any plans at all.

ragraph

It is worth mentioning that this approach requires delicate handling of projections. As elaborated in the section 3.3.1 on query parsing, projections come up a lot in decorrelating queries. Joins separated by projections can not trivially be mixed with the above method as moving projections would potentially break the rule that we demand that there is no naming conflict between columns. Even though as things stand this case is not handled our method of uniquifying columns is powerful enough that the non-conflict rule could potentially be relaxed by applying methods similar to the ones we deploy in disambiguating columns section on Query Normal forms (QNF) 3.4.

The fundamental forest structure of literally collecting each logical plan individually is highly redundant. Therefore, we abstract away the redundancy by representing the forest as a bipartite tree.

- One kind of node represents a plan fragment, or a *sub-tree*
- The other kind represents a forest of semantically equivalent subtrees earmarked with a unique hash for them. The fact that the subtrees are semantically equivalent means that they essentially evaluate to the same data. They therefore have the same normal form as discussed in section 3.4 on Query Normal Form (QNF). Finding the normal form hash of one would mean the normal form hash of the rest. This will prove useful when inserting this structure in the graph. We refer to these as *sub-forests*.

In other words, the relational algebra tree is interleaved with OR-nodes of different representations of the subquery. For example $\gamma(\sigma_{a=b \land b=c}(A \times B \times C))$ becomes $\gamma\{A \bowtie_{a=b} B) \bowtie_{b=c} C, A \bowtie_{a=b} (B \bowtie_{b=c} C)\}$. This way we mitigate the combinatorial explosion of interleaved nesting joins with other operations.

The algorithm is also described in listing 3.3.

```
def possible_joins(q):
    # Split a query that is equvanent to
    # sel(p1 and p2 and p3 and  ... , prod(q1, q2, q3, ... ))
    # into
    # [p1,p2,p3,..], [q1,q2,q3, ... ]
    and_props,subqueries =  as_prod(q)
    # If there are no top level joins traverse the query
    if len(and_props) == 0:
        return q.map_children(possible_joins)
    # All possible partitions where the partitions are non-empty
    for lqs,rqs in possible_partitions(subqueries):
        # has_free_vars(p,[q1,q2,q3 ... ]) checks if all the atoms in p
        # are either literals or columns in one of the q1,q2, ...
        lps = filter(lambda p: not has_free_vars(p,lqs), and_props)
        rps = filter(lambda p: not has_free_vars(p,rqs), and_props)
        # The props thet connect the partitions into a join.
        jps = filter(lambda p: has_free_vars(p,lqs)
                     and has_free_vars(p,rqs),
                     and_props)
        # If there are no predicates connecting it is product. We
        # disallow products.
        if len(jps) == 0: continue
        # A valid join to be considered while planning!
        lq = sel(lps,prod(rqs))
        rq = sel(rps,prod(rqs))
        yield join(jps,possible_joins(lq), possible_joins(rq))
```

Listing 3.3: Pseudo-python description of finding all possible for clarity, it is abbreviated to omit sanity checking, memoization, some type conversions, etc.

It is worth stressing that, because many of the subtrees generated this way will be equivalent. As analyzed in [64] and discussed in section 3.4, in pure functional settings like our, DAGs can have explosive complexity during traversal. Even worse in this case even the space complexity can indeed be needlessly exponential. For that reason we maintain a hash map matching hashes with the corresponding matching subtrees. Before inserting a newly created forest into our structure we perform a lookup in the hash map. If we find something we throw away the newly created forest for the optimizer runtime's garbage collector to recycle, and we use the one in the hash map instead. Otherwise, we insert our forest both in the hashmap and in the tree. This way we deduplicate our data structure to some degree. Furthermore, in the same vain as [64] we earmark all our forests with their hash so we don't have to traverse them again to compute it. The result of this is that in total we completely traverse the forest once, in order to compute the hashes of each sub-forest and then we use those hashes to test sub-forests for equality.

3.4 Query Normal Form (QNF)

It is imperative that the QDAG has as little redundancy as possible, two subqueries that describe the the same relation should be represented as the same node. Equivalence of

RA expressions is infamously NP-complete [1] but we make a best effort to normalize queries such that they are hashable and comparable via simple structural equality. The core of normalization is transforming the queries to Cartesian normal form

$$\zeta_{x_1}(A) \bowtie_{p_1} \zeta_{x_2}(B) \bowtie_{p_2} \zeta_{x_3}(\sigma_{p_3}(C)) \rightarrow \zeta_{x1;x2;x3}(\sigma_{p1 \wedge p2 \wedge p3}(A \times B \times C))$$

where $\zeta \in \{\pi, \gamma\}$.

3.4.1 QNF Structure

FluiDB's Query Normal Forms are highly polymorphic data structures separate from RA data structures. The general QNF form can encapsulate one of several special forms, all of which strive to abstract away as many valid RA rewrites as possible, and ideally achieve a one-to-one correspondence between a QNF and a table. While the QNF in its general form (listing 3.4) is specialized to represent several different forms internally to the QNF subsystem, outside of it there are two main objects:

- The normal form of a relational algebra query
- A column of a normal-form query.

```haskell
data QNFQuerySPDCF sel_f prod_f dbg_f col_f f e s =
  QNFQuery
  { qnfColumns :: col_f (QNFProj f e s) (QNFAggr f e s)
    -- ^ What columns kept. Each of these is a column on one of the
    -- QNFProd.
    ,qnfSel :: sel_f (QNFSel e s)
    -- ^ And conjuction of selection. The QNFNames here lack a qnfProd
    -- and qnfSel because they are names of this QNFQuery itself so to
    -- avoid a recursive structure that we need to keep up to date we
    -- just promise that they will be empty.
    ,qnfProd :: prod_f (QNFProd e s)
    -- ^ QNFProd e s refers to multiple rewrites of a single
    -- relation. Essentially {Query (..) (QNFQuery e s)}. Each element
    -- of this set needs to have the same set of QNFQuery's. The binary
    -- operators allowed here MUST expose plans from ONE side. Otherwise
    -- the qnfColumns will be ambiguous. In practice only Product/Join
    -- expose both sides but if there are more in the future know that
    -- qnf machinery will break.
    ,qnfOrigDEBUG' :: dbg_f (Query e s)
    ,qnfHash :: QNFKey e s
    -- ^ Cache the hash. Hashing is done separately for each field so we
    -- don't recompute hashes of unchanged fields after each operation.
  }
```

Listing 3.4: The QNF datastructure.

In section 3.4.2 on building QNFs, we will look at several specializations that are useful as intermediate. In this section we will focus on each of the fields of the `QNFQuery` constructor.

QNF Product collection

The `qnfProd :: prod_f (QNFProd e s)` field contains the unordered collection of subqueries $\{A_1, ..., A_n\}$ in the equivalent $\pi\sigma(A_1 \times ... \times A_n)$, typically as a hash-set (via the instantiation of `prod\_f` as `HashSet`). Each of the A_i terms may be one of the following:

- Primary tables
- QNFs with an opposite kind `qnfCol` (if Q is a projection an A_i may be an aggregation)
- A set of equivalent RA expressions that are not reducible to the $\pi\sigma(A_1 \times ... \times A_n$, for example $\{A \ltimes B, B \rtimes A\}$ or $\{l_N(A)\}$

Focusing on the latter case, there are certain equivalent rewrites that are not easily expressible in the QNF. For that reason instead of perpetually extending the QNF system we provide this expensive method of simply enumerating equivalent queries in RA form. These RA queries have at the leaves of the expressions either QNFs or primary tables. The only restriction that we place on these queries is that the schema of the inputs is simply forwarded to the output without modification of the individual fields so that the `cnfCol` vector can refer directly to them.

QNF Columns, projections and aggregations

Whenever a `QNFQuery` is found outside of the QNF sub-module `col_f` will be instantiated as `Either`, which is to say that a query may be an aggregation (`Right`) or a projection (`Left`). A query with no explicit projection operand at its root is translated to QNF by enumerating the columns into this field. In all of these storing them in a container of parametric type `f`, which is the main distinguishing factor between QNF columns and QNF queries: a query projection exposes an unordered bag of columns through the projection/aggregation field (`f` is instantiated to `HashBag`) while a QNF column exposes *exactly one* column (`f` is instantiated to `Identity`).

We extend the notion of a column to to define names (see listing 3.5), the atoms of predicates and other experessions (see section **??**). A name in the context of QNFs can be one of three different kinds:

- **A column of a nonprimary relation.** This relation *must* be a member of the `cnfProduct` collection of the QNF structure where the name appears. Because there may be more than one equivalent query in the product collection, we index the columns with an integer.
- **A column of a primary table**. These columns have a name and are associated with a table (`s`) which needs to be in the product collection like in the case of nonprimary columns. It is indexed much like the nonprimary relation.
- **A literal value**

```haskell
data QNFNameSPC sel_f prod_f col_f e s =
  PrimaryCol e s Int
  | Column (QNFColSPC sel_f prod_f col_f e s) Int
  -- ^ a a column c that refers to a:
  --
  -- QNF[P[c],S[c],Prod[Query[a ~ QNF]]]
  | NonSymbolName e
  -- ^ A literal.
  -- A column is a QNF that has one element in the projection set
  -- (Identity) and no debug query.

type QNFColSPC s p c = QNFQuerySPDCF s p EmptyF c Identity
```

Listing 3.5: A QNF name may be an unnamed column of a relation, a named column of a primary table or a literal.

In listing 3.6 we present the definition of the projection field of the QNF structure. It departs from the definition we examined in section 3.1 where we described the RA in two important ways, a) no names are provided for the columns and b) the columns are unordered, depending on the definition of `f`. We maintain, however, the definition of each column as an `Expr`. FluiDB makes no attempt to normalize the expressions, so equivalent expressions that differ syntactically will produce different QNFs.

```haskell
type QNFProj f e s = f (Expr (QNFName e s))
```

Listing 3.6: A QNF projection field is a collection of expressions that refer to QNF names. The particular structure of this collection is parametric. When the collection `Identity` the QNF query is essentially just a column. A normal QNF query would instantiate `f` to `HashBag`, an unordered multiset.

As mentioned, the columns in the QNF name, both primary and of general relations, are indexed in the case that there are more than one equivalent. This is an edge case that can cause problems because different indexing can cause equivalent nodes to have different QNF representations. For example, the query `select * from A as A1, A as A2 where A1.a = A2.a` is symmetric in all respects and therefore the way QNF names representing columns `A1.a` and `A2.a` are indexed makes no difference provided that the equality expression `A1.a = A2.a` is ordered consistently. Consider, however, the query `select * from A as A1, A as A2 where A1.a = A2.a + 1` which is not symmetrical.

Indexing `A1` and `A2` differently does not change the semantics of the query, but it changes the QNF representation. This is not commonly a problem, a rare edge case rather, but it is important to keep in mind. In an attempt to mitigate this we *all* different combinations of indexing are emitted by the QNF builder (see section 3.4.2).

From the perspective of QNFs, aggregations (defined in listing 3.7) are very similar to projections. However, they only operate on the non-recursive `Aggr` algebra, in contrast to the `QGroup` unary RA operator that incorporates `Expr (Aggr (Expr e))` typed expressions.

QNF moves the inner and outer `Expr` to one level up and one level down respectively, so an aggregation $\gamma_{l \mapsto f(g(h(x)))} S$ would be rewritten to the normalized $\pi_{l \mapsto f(l_1)} \gamma_{l_1 \mapsto g(l_0)} \pi_{l_0 \mapsto h(x)} S$.

```haskell
type QNFAggr f e s =
(f (Aggr (QNFName e s)),HS.HashSet (Expr (QNFName e s)))
```

Listing 3.7: The QNF aggregation form of the projection field is similar to projection only, much like the `QGroup` constructor, it also includes a `HaskSet` of exprssions on which to group.

QNF Selection

The `qnfSel :: sel_f (QNFSel e s)` field contains an expression of the selection predicate (see listing 3.8). With very few exceptions `sel_f` is instantiated as an unoredered hash containing the terms of the Cartesian normal form of the selection. For example a `qnfSel` with value $\{p_0, p_1, p_2\}$ corresponds to the selection predicate $p_0 \wedge p_1 \wedge p2$.

```haskell
type QNFSel e s = Prop (Rel (Expr (QNFSelName e s)))
```

Listing 3.8: Selection name refers to a version of the current QNF that has all fields erased except the projection.

We went into detail in the previous section how the atoms of the predicates in QNFs, when they refer to columns, take the form of QNF structures themselves. Further, we asserted that the columns in the expressions of projection/aggregations refer to the product. Similarly, selection names refer to columns of the product. This means that in the QNF of the expression $\sigma_{f(a)}(Q_0 \times Q_2 \times ...)$ the atom a is encoded as a column of one of the Q_i subqueries.

This way we can update the fields of the QNF without needing to keep the columns in sync with the columns referenced in `qnfSel`. This eliminates the need for both indexing the columns and for dealing with primary colmns in `QNFSelName`. This has to do with the way selections are being created: the RA expression $\sigma_p A$ is translated by first translating A to a named QNF. Then p refers directly to the columns of the query.

QNF Misc fields and hashing

There are two more fields in the generic QNF datastructure we described. `qnfOrigDEBUG' :: dbg_f (Query e s)` is the more straightforward one. It simply keeps track of the query used to create the QNF for debugging purposes and is disregarded by both the equality and hashing operations. Typically `dbg_f` is instantiated to `EmptyF`, an empty container to save on memory.

The other field is the `qnfHash :: QNFKey` (see listing 3.9). Because the QNF is a highly self referential data structure equality in a purely functional environment like Haskell traversing it for the purposes of equality or is a costly operation. For that reason, we depend on hashes to detect equivalence, and we cache those hashes to avoid recomputing them, leaning on the purity of Haskell. As the QNF has three distinct parts that are being often updated during building of a QNF `qnfHash` is represented as a tuple of their respective hashes. This way, when one of the fields is updated, we do not have to recompute the hash of each of the others.

```haskell
type QNFKey e s = (Int,Int,Int)
```

Listing 3.9: A key that uniquely idnetifies a QNF is made of three separate hashes, one for each part of the qnf structure so that they can be updated independently.

Since QNFs are rarely used as anything other than a token for relating RA form queries FluiDB saves on memory and use `QNFKey` in place of the `QNFQuery` structure (see listing 3.10). On the other hand, in a more pedantic but correct mode, we can avoid depending on the quality of the hash function and define equality as recursively checking between all types (see listing 3.11). In practice, we have never found the former method to yield a false positive for equality.

```haskell
instance Eq (QNFQuerySPDCF sel_f prod_f d col_f f e s) where
  x == y = qnfHash x == qnfHash y
```

Listing 3.10: A fast and loose definition of equality between QNFs that depends on the quality of equality.

```haskell
instance Eq (QNFQuerySPDCF sel_f prod_f d col_f f e s) where
  x == y = qnfProd x == qnfProd y
    && qnfSel x == qnfSel y
    && qnfColumns x == qnfColumns y
```

Listing 3.11: A very inefficient but correct equality between QNFs.

3.4.2 QNF Computation

In this subsection we will look at a few parts of the QNF building process that are maybe less than obvious.

QNFs are built bottom up from the leaves to the top. Building the QNF we apply the operations incrementally to a named QNFs (NQNF). Applyig the operator σ_p to a QNF A we need to be able to to relate the symbols appearing in p with the columns of the QNF A. To mitigate this define a *named QNF* which is simply a tuple of a QNF along with a mapping between symbols and columns of that QNF. For example, the name map in the NQNF derived from the query $\pi_{n \mapsto f(a)} A$ would map the symbol n to a column with the elements $\{f(a) | a \in A\}$.

The entire process of computing QNFs works within a monad stack with the following features:

- A `ListT` monad allows for non-deterministic computation in order to account for the different possible column index assignments described in subsection 3.4.1 on QNF projections/aggregation.
- Since the resulting QNF is a highly self referential structure, it is to be expected that a lot of the computation might be duplicated. The cache object is also shared between different QNF building processes that we know are likely to have have a lot of overlap, for example, when inserting nodes into the graph without shared caching FluiDB would be repeatedly computing the QNF of the same RA subqueries.
- The process of building a QNF can fail when processing malformed RA forms.

These characteristics are encoded in the monad within which we build the QNF (listing 3.12)

```
-- | The monad in which the computation happens. Not all transformations
-- require non-determninsm.
type QNFBuild0 e s = StateT (QNFCache e s) (Either (QNFError e s))
type QNFBuild e s = ListT (QNFBuild0 e s)
```

Listing 3.12: QNF computation monad provides non-determinism, caching, and error handling.

Every QNF building function accepts one or two NQNFQuery forms and an operator unary or binary relational algebra operator to be applied to the query. It returns an `NQNFResult` (listing 3.13) contains

- A named QNF which is the main payload of the QNF building function.

- A mapping between input and output QNF columns. When there is no direct mapping (like in the case of aggregations) this is an empty list.

- The RA operator that was provided as input with the expression atoms translated to QNF names refering to the input or output columns. The way the names in the operator expressions are translated to `QNFName`s is specific to each operator. For example, in the selection operator, all atoms are directly translated using the name map of the input `NQNFQuery`. When the operator is a projection $\pi_{\{n_0 \mapsto K_0, n_1 \mapsto K_1, ...\}}$, the builder function takes as input a QNF Q_{IN} and the parameters of the projection $\{n_0 \mapsto K_0, n_1 \mapsto K_1, ...\}$, and outputs a `QNFResult` containing the QNF Q_{OUT}. In the projection operator output, the left hand side atoms of the projection $\{n_0, n_1, ...\}$ are columns of Q_{OUT} and the atoms in the right hand side expressions $Q_0, Q_1, ...$ are columns of Q_{IN}. This is for convenience assuming that the RA query that was just normalized is a logical plan. The top level operator can then be used to annotate QDAG elements that relate to the normalized query. The importance of this is analyzed in depth in section on symbol correspondence.

```haskell
data NQNFResultDF d f op e s =
NQNFResult
{ nqnfResNQNF :: NQNFQueryDCF d Either f e s
  ,nqnfResOrig :: op
  -- Map the names exposed by the input qnf to names on the output
  -- qnf. Unless qnfResOp is QProj, QGroup each input name corresponds
  -- to 0 or 1 name in the output. So if qnfResOp is QProj or
  -- qnfResInOutNames has no meaning and will be `const Nothing`
  ,nqnfResInOutNames :: [((e,QNFCol e s),(e,QNFCol e s))]
}
```

Listing 3.13: The internal QNF building functions provide some more information that was created during the generation of the QNF, precisely a name map relating column names to QNF names, a map relating input QNF names to output QNF names, and the top level operator with the names translated appropriately to input or output QNF names.

The details of most QNF functions are fairly mundane and easy to guess with the possible exception of products. When calculating the the product of two NQNFs Q_1 and

Q_2 we first find conflicts, where columns of Q_1 can be found in the product collection of Q_2. If none are found, we can simply merge all of their fields and recalculate the hashes. If there is at least one conflicting pair of `QNFName`s (c_1, c_2), non deterministically increment the index of c_1 or c_2. For example, if we want to compute the QNF of the product $\sigma_{a_0=1}(A_0) \times \sigma_{a_0=2}(A_0)$ we get in QNF-like form $\sigma_{a_0=1 \wedge a_1=2}(A_0 \times A_1)$ and also $\sigma_{a_1=1 \wedge a_0=2}(A_0 \times A_1)$.

```
def qnf_product_code(nqnf1, nqnf2):
    # The named qnf is a pair of a map names→columns and a QNF.
    nm1, qnf1 = nqnf1
    nm2, qnf2 = nqnf2
    # Find relations in the product set that could cause name
    # conflics.
    conflicts = [c for c in qnf1.qnfProd if c in qnf2.qnfProd]
    for c in conflicts:
        # Python does not support non-determinism but with ListT we
        # can do it in haskell. The `nondet' function non
        # deterministically selects one of the arguments and returns
        # it. The result of the overall computation is a stream of all
        # possible results.
        #
        # The `increment_indices function returns the same named CNF
        # with all columns' indeces that corresond of the argument CNFProd
        # incremented.
        ncnf1, ncnf2 = nondet(
            (increment_indices(ncnf1, c),ncnf2)
            (ncnf1,increment_indicse(ncnf2, c)))
    return ncnf1 + ncnf2
```

Listing 3.14: The qnf product.

3.4.3 Pragmatism of QNFs

As mentioned, query equivalence is an NP-complete problem. QNF is only a heuristic that does fail to identify many cases of equivalent RA queries. For example, our design of QNFs does not capture the equivalence $\sigma_p A \cup \sigma_{\neg p} A = A$. It captures enough rewrite rules, however to be useful in most common cases. In particular, it covers the following RA properties:

- $\sigma_p \sigma_q A = \sigma_{p \wedge q} A$ - $\pi_m \pi_n A = \pi_{m \circ n} A$ - $A \bowtie B = B \bowtie A$ - $(A \bowtie (B \bowtie C) = (A \bowtie B) \bowtie C$ - $\sigma_\theta (A \times B) = A \bowtie_\theta B$ - $\pi_m \gamma_{n,g} A = \gamma_{m \circ n, g} A$ - $\gamma_{m,g} \pi_n A = \gamma_{m \circ n, g \circ n} A$ - $A \not\ltimes B = B \not\rtimes A$* - $A \cup B = B \cup A$ *

3.4.4 Symbol correspondence

We need to normalize the names in each operator, but we also require that there is no naming ambiguity at any level. Why, then, is it necessary to normalize the symbols if they are unambiguous in each context? If we were only dealing with one query we could declare that we trust that the user would be consistent when naming their columns (`select col as name where ...`), declare that the broader problem of RA equivalence is beyond the scope of this thesis and move on to more central issues to FluiDB. It is, however, the case that the user is not in total control of naming columns. When exposing unique ids and during the decorrelation transformations, FluiDB produces unique column names stochastically. This is not a problem until different queries interact in the context of the graph.

Normalizing the queries that correspond to n-nodes is of little use if we do not also normalize the symbols that refer to the columns of each n-node. To produce the code implementing an operator we need to be able to establish a stable correspondence between

- the symbols that refer to the columns of the input tables the symbols that refer columns that refer to the output the symbols in
- the predicates and expressions of the RA operators that t-nodes correspond to.

This is a problem unique to FluiDB because most systems will produce variants of QDAGs that are specific to the query being planned. FluiDB maintains a QDAG that must maintain its consistency between different queries.

The first step before inserting an operator in the graph is to normalize the symbol name. We consider that two symbols are equal only if they refer to the same column. This is done by means of converting the query into normal form as described previously.

This is required to resolve the possible ambiguity of symbols with the same name but different scope when expressions are moved around. It abstracts away the symbol names but creates some complications. For example, in the query $\sigma_{a=x \vee a=z}(\gamma(\sigma_{a<y}(A))$ the a in $a = x$ and the a in $a < y$ are not equal symbols as they refer to different columns. The former refers to a column in the table $\gamma(\sigma_{a<y}(A))$ and the latter in the table A. They both refer to some transformation of a column labelled a. On the other hand a in $a = z$ and a in $a = x$ are indeed equivalent as they refer to the same column. To clear up the confusion we will say that symbols are *strictly-equivalent* if they refer to the same column but that they are *provenance-equivalent* when they refer to different columns but they are transformations of the same column, i.e., they would be represented by the same in a normal relational algebraic setting.

In particular, each materialized n-node must have a precise schema that represents the order of columns. The identities of these columns must be agnostic to the particular names decided by the user, the decorellating subsystem, or any other query processor. Otherwise it would be impossible for two clusters to share an input/output n-node would both be able to interact with it only in the case that they happen to be in total agreement about the column names of the relation that the n-node represents.

Thus, to interact with the nodes inserted in the graph previously and to allow future queries to match the nodes we insert we split it's symbols' representation in two notions.

- A local notion of a symbol is subject to the names given by the circumstances of its creation (user defined names,generated symbols,etc)
- a global notion that is a symbol representation dependent only in the column itself.

The local symbol is useful to find provenance equality should it be needed (being cautious when merging symbols based on them sharing a local notion that the global notion maintains correctness, i.e., it refers to the right column). It is clear that local notions of symbols are sensitive to the name chosen, the global is not.

Using the global notion of symbols creates a problem with the correspondance of the query schema of the output with the schemata on the input side. In other words, if we throw away the names of the columns in the inserted operations in favor of column unique ids, how would the code generator know how to arrange the elements of each input tuple it encounters to generate the output tuple?

To mitigate that, we use the internal operations of the QNF query builder. To build a cluster we use as parameters a named normal form (a query in normal form and a mapping between the column names and column ids, along with an n-node for each input and output relation, and the operator with column references in terms of local notions of symbols. As described in detail in section 3.4.2 on building QNF, the result of building the QNF that corresponds to an n-node also returns the root operator with globalized symbols, as well as a correspondence between globalized symbols at the input with globalized symbols at the output side of operator.

Each cluster is marked with all the relevant information required in order for the useful to the code generator. But is that enough? Not quite. As we have alluded to, each relation we deal with, and therefore each n-node, must be associated with a schema which contains information about:

- Which columns are available
- What is the type of each column
- Which columns are constant values
- What sets of columns form unique sub-tuples
- What is the order in which the columns are store.

3.5 Query clusters

Clusters are the highest level building blocks of the graph. This section describes a very high level of the semantic role of the cluster as a set of connected n-nodes and T-nodes.

Clusters represent *logical operations* and they are the connection between the basic graph and the relational algebra semantics.

3.5.1 Cluster polymorphism

As alluded to when describing the QDAG in section 3.2, clusters come in different shapes depending on what the underlying operator is. In this section we will take a closer look at the structure of the different cluster types. Each type of cluster has input, output and intermediate slots, and each of the slots typically contains an n-node reference and an RA operator that relates the node to the cluster. They also contain slots to accomodate the t-nodes connecting said n-nodes. In dealing with clusters it is assumed that input and output n-nodes contain values that may be shared by other clusters. In contrast, intermediate n-nodes and t-node slots are always completely private to each cluster.

We define are three different kinds of clusters and a unified type `AnyCluster`:

- **`NClust`** is a cluster containing exactly one slot which corresponds to a primary table.

- **`UnClust`** a cluster corresponding to a unary operation. It has two slots in the output position, one slot in input position, no intermediate n-nodes and a single t-node slot 3.6.

- **`BinClust`** is a cluster corresponding to a non-join binary RA operation. It has one output slot, two input n-node slots and a t-node slot 3.7.

- **`JoinClust`** augments the **`BinClust`** with two intermediate n-nodes, two extra output n-nodes, and two input t-nodes to facilitate the antijoin results of a FluiDB join operation 3.8.

Every cluster must always be hashable and the hash is cached inside the cluster itself.

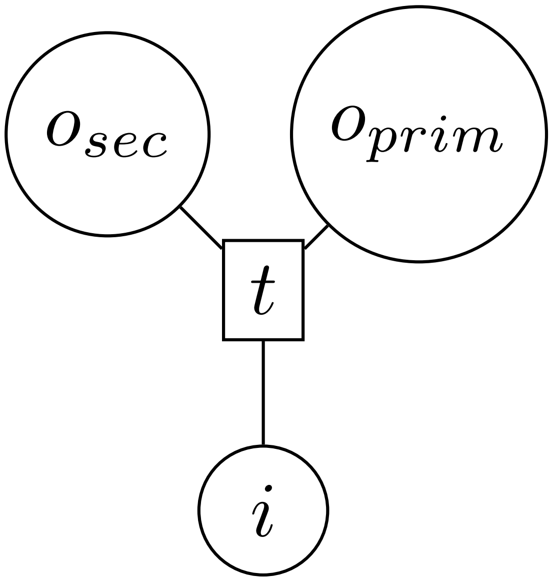


Figure 3.6: The structure of a **`UnClust`**.

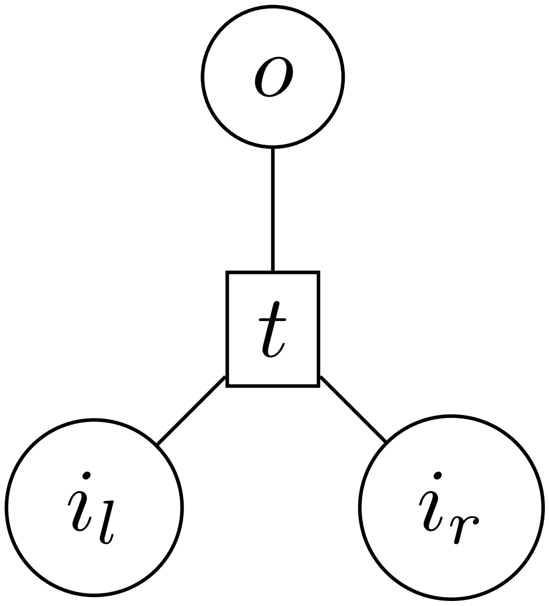


Figure 3.7: The structure of a `BinClust`.

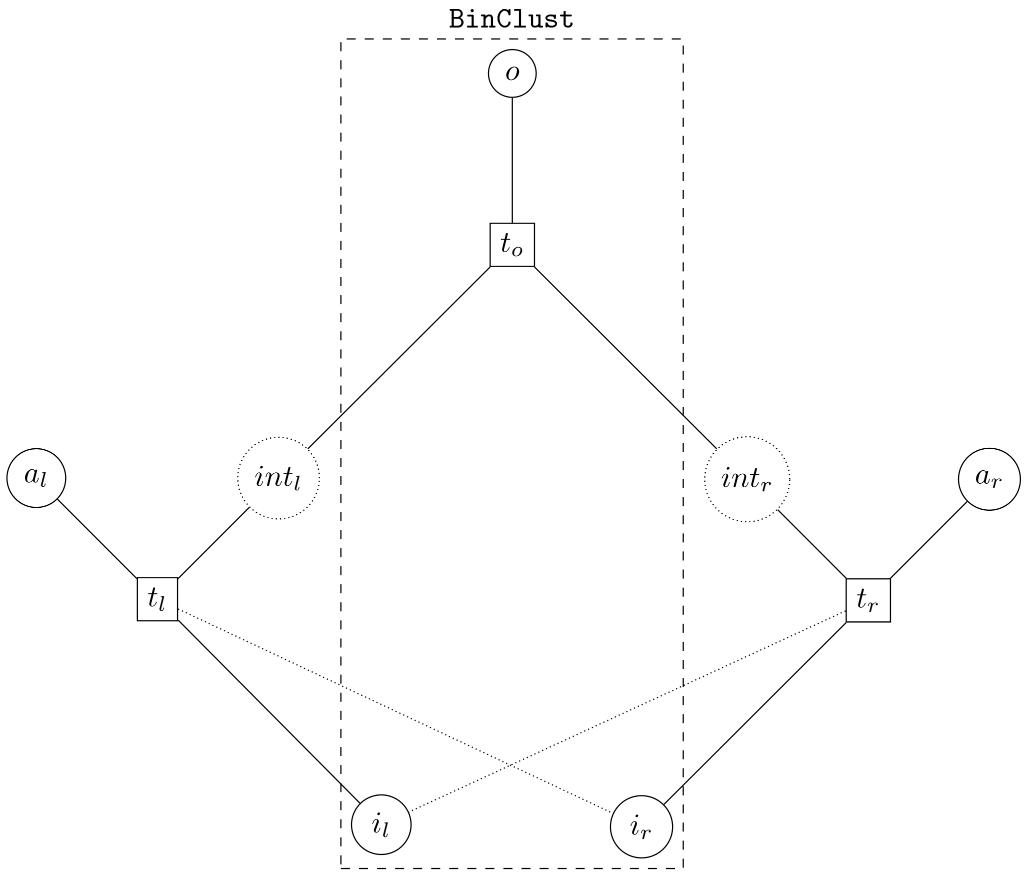


Figure 3.8: The structure of a **`JoinClust`** notice how it contains a **`BinClust`**.

3.5.2 Populating the graph

Once the query is transformed into a forest of possible join orderings as described in section 3.3.3 on enumeration of possible joins, the query is transformed into a forest (listing 3.15) of plans we can insert it into the tree. This is an iterative process where each operation in each of the trees in the forest is translated to a cluster which is then inserted into the QDAG. The process if inserting a sub-forest or subtree into the QDAG has the effect of extending the QDAG by the connected clusters corresponding to the operators at the nodes of the forest, and also returns an n-node that corresponds to the subforest or sub-tree relation. However, since the insertion function is idempotent, i.e., inserting the same forest in the QDAG more than once has no effect either on the returned value or on the resulting QDAG, we memoize the function. Since the same function recursively inserts all sub-trees and sub-forests we get for "free" a mechanism of not traversing the same tree twice during insertion.

```haskell
data QueryForest e s =
  QueryForest
  { qfHash :: Int
    ,qfQueries --
    :: Either --
    (NEL.NonEmpty (Query e (QueryForest e s))) -- An actual forest
    (s,RelationShape e s) -- A final leaf
  }
```

Listing 3.15: The definition of the query forest. The query forest is hashed so that we can avoid traversing the same query forest repeatedly. The query forest is essentially a non-empty of queries with forests at their leafs.

The process of traversing the forest and inserting the clusters in the graph is fairly straightforward. The actual creation of the clusters, however, is slightly more convoluted. First, a cluster of a type corresponding to the query operation at the forest node is created (see section 3.5.1 on cluster polymorphism). That cluster initially contains QNF forms at its edges. The QNF forms are then matched against existing QNFs in the QDAG and the missing ones are generated. If no new nodes were generated and the matched nodes all correspond to the same existing cluster, we infer that the cluster is already in the QDAG and the process finishes . Otherwise, we need to create and insert a cluster and a corresponding propagator (see section **??**). Finally all new nodes, QNFs, and the cluster are registered and cross matched so they are easily cross referenced. We memoize the entire process using the `qfHash` to avoid re-traversing the same path.

As an example of what a more realistic QDAG looks like, first run through all 12 queries of the SSB TPC-H benchmark (see chapter 7) produces the graph presented in figure 3.9.

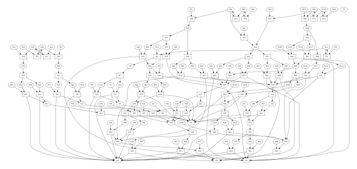


Figure 3.9: A QDAG accumulated out of every node in the SSB TPC-H.

3.6 Relation shapes

In the section on QNFs we discussed the need to abstract away information that pertains to the particular structure of a relation in favor of semantic information that is rewrite and data independent. Relation shapes cover the opposite need: they encode the particular cardinality and shape of a relation at a specific point in time.

A relation shape is a datastructure that contains information about the shape of the result of query.

```haskell
data RelationShape' e' =
  RelationShape
  { rsSchema  :: [(e',ColumnProps)]
    ,rsUnique  :: NEL.NonEmpty (NEL.NonEmpty e')
    ,rsSize  :: RelationSize
  }

type RelationShape e s = RelationShape' (ShapeSym e s)

data ShapeSym e s =
  ShapeSym { planSymQnfName  :: QNFName e s
    ,planSymQnfOriginal  :: e
  }
  deriving (Show,Generic)
```

A shape contains all combinations of columns that comprise a unique subtuple. This is useful, among other things, to determine foreign key joins. The expression

$$Q_1 := A \bowtie_{bId_A = id_B \wedge cId_A = id_C} (B \times C)$$

behaves like a foreign key join while the join

$$Q_2 := A \bowtie_{bId_A = id_B} (B \times C)$$

does not because for the expression$B \times C$ neither id_B nor id_C are unique on their own. Their combination is. Therefore, the cardinality $|Q_1|$ is at most the size of A because each row of A matches at most one row of $B \times C$.

So the unique subtuples are computed using the following rules:

- In general the set of unique subtuples of $A \bowtie B$ is computed by concatenating all combinations of the unique subtuples of A and B.
- The unique subtuples of a selection, semijoin, antijoin, and limit are the same as the unique subtuples of the underlying relation.
- The unique subtuples of a projection are the unique subtuples of the input that are entirely exposed. The reader is reminded that FluiDB's notion of RA requires that there is at least one such unique subtuple.

3.6.1 Shape propagators

Shape propagators are based on the the idea of the propagator as first introduced in [25] and analyzed more in depth in [65]. Very briefly, a propagator network is a hypergraph of cells containing mutable state and hyperedges as multi-directional functions that, when triggered, update the state of the connected cells so they are consistent with each other. The state of each cell represents some information about a value (like a bound or a value paired with a certainty metric), and the propagator takes into account the state of each cell and propagates that information to the other cells (for example tightening their bound or finding a linear combination of alternative values based on the certainty of each). Kmett [74] formalized the notion of "information about a value" drawing from [38] encoding it as a lattice with $\top$ meaning contradiction, and $\bot$ meaning that there is no information about the value. Then the new values of cells are being joined ($\vee$) with the old ones and the value is updated to reflect the combination of information. The example provided by Kmett is cell being embedded in a 3 level lattice where the bottom level is `Nothing`, the middle level is `Just x` and the top level being a Haskell error (ironically denoted as $\bot$ in literature unrelated to propagators).

As the name of our propagator implementation suggests, our idea is to correspond each cluster to a propagator to form a propagator network that computes the shapes of the relations corresponding to n-nodes.

Intuitively, we model a propagator as a partial function that changes the values at the edges of a cluster. Each cluster is equipped with a propagator which, when triggered synchronizes the values at the edges of the cluster. The partiality of the function stems from the fact that it might detect irreconcilable inconsistencies between the cells (see listing 3.16). This conception of the propagator departs slightly form the conception described in the paper in that it clearly separates the *cluster* as a datastructure that holds a fixed number of interdependent values, and the function that updates the values in that cluster.

```
type ACPropagator a e s t n =
  EndoE e s (PropCluster a NodeRef e s t n)
type EndoE e s x = x -> Either (AShowStr e s) x
type ShapeCluster f e s t n =
  PropCluster (RelationShape e s) f e s t n
type PropCluster a f e s t n =
  AnyCluster' (ShapeSym e s) (WMetaD (Defaulting a) f) t n
newtype WMetaD a f b = WMetaD { unMetaD :: (a,f b)}
```

Listing 3.16: A propagator matches a cluster with shapes at the edges to the same kind of cluster with the shapes synchronized.

Given a cluster with n-nodes at the edges, we lookup the shape of each node and add that to the cluster edge. Once a cluster is triggered and the values synchronized those values are checked against the old ones and for the ones that were updated we find the clusters in which they participate and run the same process. The convergence of this process is justified in [38].

3.6.2 Defaulting functor

While discussing propagators, we mentioned that the cells of a propagator involve more structure than raw values. Indeed, FluiDB wraps these values in a functor that we dub as *defaulting functor* to reason about the consistency of the relation shape. In particular we require that the cell can handle cases where

- The value for the shape is not computed yet
- The shape is inferred via the propagator network and is therefore subject to change and
- The shape stored in the cell corresponds to a materialized node and other parts of the FluiDB's internal state depend on it.

The latter point is salient. Different logical plans can lead to different, yet semantically equivalent as per the FluiDB RA.

The defaulting functor (DF) may hold up to two values (see listing 3.17).

- An empty DF means we have no information about the value.
- A single value (the "default" value) corresponds to a derived shape that was computed using the propagator network. The default value is updated when propagators are triggered.
- A full value exists alongside a default value and is constant through its lifetime as it matches the value of some state that is external to the propagator network, in our particular application it represents the shape of a materialized relation. Until that relation is deleted, and the full propagator is demoted to a defaulting one, the registered shape is not allowed to change.

```haskell
data Defaulting a =
  DefaultingEmpty
  | DefaultingDef a
  | DefaultingFull a a
```

Listing 3.17: The defaulting functor definition.

Since instances of the DF are meant to accommodate propagator values, adhering to Kmett's approach to cell value maintenance, we define a semilattice over the DF and define it in terms of a monoid (listing 3.18).

Semigroups are not commutative, so we give meaning to the order. The left operand of the semigroup operator is the newer value and the right one is the older value. Therefore, the DF does not require all the properties of the semilattice which is required to commutative and associative. In general, the semigroup implemented by the underlying type of the DF is taken to mean `better <> backup`, i.e. keep as much of the left operand as possible unless the right operand is found to be "better" for some definition of better. For example, when calculating the cardinality we include a value of certainty between $[0, 1]$. A more certain right-hand cardinality is preferred over a less certain left-hand side.

Bringing our attention to the implementation of the `Def <> Full` combination in the semigroup definition 3.18: it inverts the order of the combination of the default values. The reason we do this is to give priority the the full valued (materialized) nodes when propagating to their immediate neighbors. We need this when committing an operator to the code generation.

```haskell
instance Semigroup a ⇒ Semigroup (Defaulting a) where
  DefaultingEmpty <> a = a
    a <> DefaultingEmpty = a
  DefaultingDef a <> DefaultingDef a' = DefaultingDef $ a <> a'
  DefaultingDef a <> DefaultingFull a' b' =
    DefaultingFull (a <> a') b'
  DefaultingFull a b <> DefaultingDef a' =
    DefaultingFull (a <> a') b
  DefaultingFull a b <> DefaultingFull a' _ =
    DefaultingFull (a' <> a) b

instance Semigroup a ⇒ Monoid (Defaulting a) where
  mempty = DefaultingEmpty
```

Listing 3.18: The join semilattice that is defined in terms of the

When committing to a query plan we are also committing to full values for the defaulting values in the propagator cells. Every operator of the plan corresponds to a relation shape propagator. When committing an operator to the plan by translating it into C++ code, we promote all the defaulting values at the edges. It is clear that the generated code expects a certain data layout and therefore the *full parts of the values must be structurally consistent with each other*.

As we went over in the previous sections, however, the default values are consistent with other default values of the same cluster w.r.t. the amount of information they hold, but not w.r.t. the order of the columns in the relation shape. Therefore, before promoting the default functors to full functors by duplicating the default value to fill both fields of the `FullDefaulting` constructor, we trigger the propagator to enforce full consistency between the defaultvalues.

The full part and the default part of the the defaulting value are not necessarily structurally synchronized. We *internally synchronise* a cell before triggering the propagator to make sure we maintain the consistency of the newly promoted cells with the already existing full cells.

It is notable that there is no formal guarantee that the propagator will be able to create structurally consistent cells. In that case the propagator will fail. Handling of this failure is beyond the current scope of this work.

When an n-node is to be materialized, the corresponding defaulting functor containing the value is *promoted*. Promotion of propagators is copying the default value of a non-full prpagator to the full state (listing 3.19).

```
promoteDefaulting :: Defaulting a → Defaulting a
promoteDefaulting = \case
  DefaultingEmpty        → DefaultingEmpty
  DefaultingDef x        → DefaultingFull x x
  d@(DefaultingFull _ _) → d
```

Listing 3.19: Promoting of defaulting functor happens during code generation when an n-node is materialized.

3.7 Summary and conclusion

In this chapter, we detailed how FluiDB processes queries. From parsing to generating a graph of queries and inferring the shapes of the relations in that graph. We also discussed different conceptions of relational algebra that are used in FluiDB (normal RA and QNF) and reverse operations. This will serve as a solid bedrock on which we can build the query planner and on top of the code generator.

There are a few shortcomings to this model that we have not yet addressed that we plan to implement in later incarnations of FluiDB. The most important ones being:

- As things stand, the QDAG will keep growing infinitely as new queries arrive and there is no obvious way to prune it while guaraneeing the materializability of all the nodes.

- There is no obvious path to supporting updates in the primary tables. There has been a lot of work on materialized view maintenance but most of it assumes that the primary tables are already materialized.

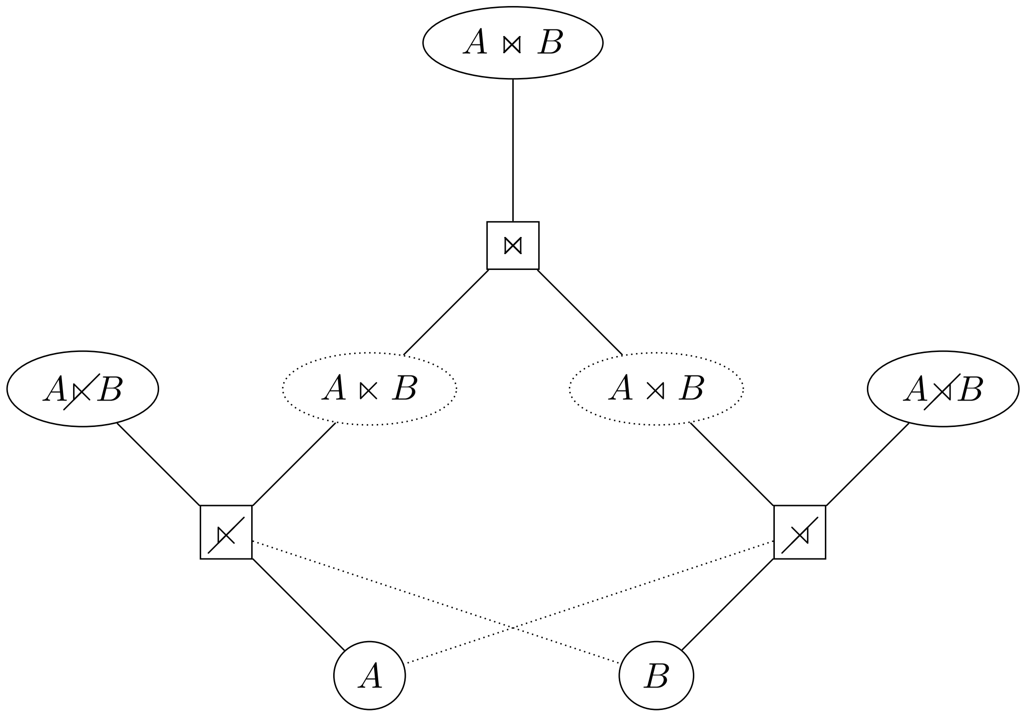


Figure 3.4: The QDAG corresponding to a join is broken down in multiple t-nodes to facilitate greater flexibility.

CHAPTER 4

Physical planning

It's time to get physical,
physical.

(Someone)

Chapter summary

- We introduce a monad-based functional method of weighted backtracking search that supports *once* and *fallback*, two cut-like operations.
- The FluiDB query planner follows an A*-like method for seaching plans plans are searched in a top-down fashion, guided by a cost model that takes into account the cost historical queries.
- The query planner can make use of *forward* or *reverse* triggering of operators.
- The garbage collector clears up space on-demand maintaining the materializability of all the nodes in the graph.

This chapter goes in detail about the architecture of the logical planing of a query. Query planni is based on an A*-like search algorithm for through the space of partial plans. We initially describe the fundamental computational structures that facilitate this search and their unique properties that allow us the necessary flexibility. Then we discuss the basic backtracking algorithm and the way the graph is traversed. We move on to discussing the garbage collection that generates plan fragments that allow the plan to free up space in the underlying storage. Finally we talk about the cost model that guides the evolution of the plan.

4.1 HCntT logic monad

The FluiDB planner is designed in terms of a backtracking search algorithm that searches in the space of subplans for an optimal (or good-enough) plan. Because the algorithm involves a lot of complicated heuristics it is important that a powerful underlying model is deployed that matches the purely functional infrastructure in which FluiDB is written. We require a monad that supports both weighted search and soft cuts. Because none of the solutions we dound in the literature support both we we developed the `HCntT` backtracking monad that is described in this section.

4.1.1 Background

Monads for backtracking in a functional context have been proposed in various incarnateions. The most common monad that encapsulates `ListT m a = forall b . (a → m b → m b) → m b → m b`, also dubbed the Church encoding of lists or an application of the Cayley theorem on list, first proposed (in an untyped format in scheme) by [3], but most authors cite [13] which seems to have redescovered an application of the concpetsin 2000, and also [73] who focused on a notion of fairness in backtracking. As shown in [66] this representation has $O(n^2)$ BFS complexity.

The other common representation of a list transformer, is the more straightforward

```
newtype ListT m a = ListT (Maybe (a,ListT m a))
```

Which mirrors precisely the idea behid MonadLogic from [73]. As demonstrated in listing 4.1 the MonadLogic able to cons(`reflect`) and uncons(`msplut`).

```haskell
class MonadPlus m ⇒ MonadLogic m where
  msplit :: m a → m (Maybe (a,m a))
  -- ... other methods are defined in terms of msplit

reflect :: Maybe (a,m a) → m a
reflect Nothing = mzero
reflect Just (a,as) = pure a <|> as

-- msplit >=> reflect == return
```

Listing 4.1: The logic monad typeclass

In [73], on top of this framework they build an `interleave` and a monadic bind (»—) operator that works with it to make computation be slightly more "fair". Interleaving is fair between two computations, when more than two are involved, half the time goes to the first computation, out of the half that is left, half ($1/4$ of the total) goes to the second, out of the $1/4$ left, half ($1/8$) goes to the third and so on. While this powerseries that describes the way the way resources are allocated to branches may seem arbitrary, in practice it may mean the difference between terminating and non-terminating computations. For example, the code in listing 4.2 can be translated to something like the code in listing 4.3 which does terminates due to the fact that while it does not give the same change to all the branches it does not completely starve any of them.

```haskell
nonTerm :: [(Int,Int,Int)]
nonTerm = do
  (a,b,c) ← (,,) <$> genNaturals <*> genNaturals <*> genNaturals
  guard $ a + b - c == 10
  return (a,b,c)
```

Listing 4.2: Using a simple list to drive non-determinism is implicitly equivalent to a DFS algorithm which in many useful cases does not terminate.

```haskell
interleaveTest :: [(Int,Int,Int)]
interleaveTest = runLogic @[] $ do
  (a,b,c) ← genNaturals >*< genNaturals >*< genNaturals
  guard $ a + b - c == 10
  return (a,b,c)
```

Listing 4.3: Interleaving (in this example `>*<`) is not *actually* fair in the sense that it does not give all the processes

So can we do better than terminating? Sometimes we can with weighted search. Weighted search refers to the backtracking search where we apply weights, or priority, to the branches. Branches with higher priority are scheduled before branches with lower priority. A sketch of the API is demonstrated in listing 4.4. The backtracking monad implements the `halt` operation (called `tell` in [66], presumably to echo the `MonadWriter` interface) which accepts a value indicating the priority of the current branch and yields control to the scheduler. The scheduler then passes control to the highest priority branch. The value passed to `halt` must implement a monoid such that the priority of a branch is the concatenation of all values passed to halt up to that point.

```haskell
-- | Non-deterministically return an integer but before that pass
-- control to the scheduler which prioritizes branches based on a
-- total sum which it tries to minimize.
stream :: (HeapKey h ~ Sum Int,IsHeap h,Monad m) ⇒ HCntT h r m Int
stream = go 0 where
  go i = do
    halt $ Sum i
    return i <|> go (i+1)

-- | Non-deterministically choose three numbers and only keep the
-- branches that satisfy a + b - c == 10. Avoid diverging by prefering
-- branches for which the sum of the numbers is minimum.
test2 :: IO [(Int,Int,Int)]
test2 = takeListT 5 $ dissolve @(CHeap (NonNeg Int)) $ once $ do
  (a,b,c) ← (,,) <$> stream <*> stream <*> stream
  guard $ a + b - c == 10
  return (a,b,c,x)
-- > test
-- [(5,5,0),(6,4,0),(5,6,1),(6,5,1),(6,6,2)]
```

Listing 4.4: Prioritise branches that we want to be executed first.

With that in mind, for the FluiDB planner we require that our logic framework supports the following features:

Weighted search: Not all plans that match our criteria, i.e. that solve the query within the space are equally admissible. We need to find as good plans as possible and we do not want the planner to spend time looking into plans that are unlikely to be efficient. For that reason neither breadth-first nor depth-first traversals are ideal for our purpose. We need a robust way to search in a weighted manner.

Soft-cut/either: As we will see in more detail in the next section, the planner is initially optimistic about being able to materialize a node until it hits the budget limit. If the budget turns out to be large enough FluiDB should completely disregard the the branch with the node should be completely forgotten about. The reason is that the later a GC happens the more options it will have and therefore the better job it will do. To achieve that we implement an operator we dub `<//>` (pronounced *eitherl*) which is similar to prolog's soft-cut.

Once: In the context of non-weighted search it is fairly straightforward to demand that a subcomputation yields no more than one value (prolog's `once`). Simply run the entire computation in-place requesting one result and if it doesn't fail return that result. This approach can still work with weighted but we would like the `halt` calls inside the computation to have global effect for the scheduler. In other words we want the scheduler to be able to interleave this branch with other branches while preserving the semantics of **once**.

None of the work we could find easily implements all the above features simuntaneous for the FluiDB planner so we implement yet another backtracking monad transformer, `HCntT`.

4.1.2 The HCntT monad

The `HCntT` monad combines the powers of delimited continuations and much like [66] it rsearches the result in layers (see listing 4.5).

```
type HCntT h r m = ContT (HRes h m r)
  (ReaderT (HeapKey v)
   (StateT (CompState h r m) m))

newtype HRes h m r = HRes (Tup2 (h (Brnch h r m)),[r])
```

```
type Brnch h r m = ReaderT (HeapKey v)
  (StateT (CompState h r m) m) (HRes h m r)
```

Listing 4.5: The `HCntT` monad transformer allows continuation based non-determinism that allows switching between branches.

Here we use the continuation monad to capture the rest of the compuation and be able to access the final result from the scope of each operator. The result is represented as `HRes` which is an intermediate layer of the continuation. Returning empty results should be passing. It contains a heap of halted branches and a list of concrete evaluated valies. The heap type is parametric to allow flexibility with respect to how to best handle the particular key types (see listing 4.6). The heap is required to be stable, i.e. items with the same key are returned in the order they were inserted. The HeapKey `mempty` must be less than (higher priority) all other `HeapKey` s.

```
class (forall v . Monoid (h v),Functor h,Monoid (HeapKey h),Ord (HeapKey h))
  ⇒ IsHeap h where
  type HeapKey h :: *

  popHeap :: h v → Maybe ((HeapKey h,v),h v)
  singletonHeap :: HeapKey h → v → h v
  maxKeyHeap :: h v → Maybe (HeapKey h)
```

Listing 4.6: We parameterize over heaps to allow the user to decide an efficient priority queue for the branches.

A halted computation branch, denoted as `Brnch` is simply a computation that evalautes to HRes. The computation must be able to interact with CompState as required by the infrastructure supporting `<//>`.

The `HCntT` on its own is not a very useful object, it needs to be *dissolved* so the values can be used. Disolution (listing 4.7) is the process of turning an HCntT value to a `ListT` value. The `ListT` will lazily produce just as many results as are required, and of course just as many effects as required, and not more. This indicates that the process of building comptuatins is composable but not incremental. The price of the `<//>` combinator that we described is that, unlike in other non-continuation based monad, once we start drawing results from the computation we can not apply further constraints on the resulting objects. Contrast that with the case of `ListT` where we can draw the first couple of results and then use the rest in a different computation.

```haskell
dissolve :: (IsHeap h,Monad m) ⇒ HCntT h r m r → ListT m r
```

Listing 4.7: Disolution is the process of turning an `HCntT` computation into a `ListT`.

The way disolution works then (listing 4.8) is to first commit to the computation constructed by applying the continuation and obtaining an `HRes`. If `HRes` provides concrete results they are yielded one by one into `ListT`. Then, if the heap is not empty, the highest priority branch is popped and scheduled to run until it yields a new `HRes`. The concrete results are yielded into `ListT` and the new heap is combined with the old one. Scheduling a branch entails running the reader layer of the monad transformer using its pervious value. As we will see, halting the branch will update that value incrementally.

We use `Tup2` (a functor that is simply `data Tup2 = Tup2 a a`) in the type of `HRes` to allow us to be flexible on whether we use BFS or DFS search among the same-priority branches. As mentioned when describing the heap, a valid heap for `HCntT` must be stable. While this is a weighted search, it remains a question of how the options that have the same priority should be ordered. If we had just one heap, appending the newly produced heaps to the left would make for a depth first traversal of the same-priority search space while appending on the right would result in a breath first traversal. We want the operators to have the flexibility to decide on which side each branch should be appended and for that reson the `HRes` actually contains two heaps: one appended to the left (DFS), and one to the right (BFS). This is of particular importance to the correct implementtion of `<//>` as we will see shortly.

```python
def dissolve(branches):
    while len(branches) > 0:
        priority,best_branch = branches.pop()
        (sub_branches_left,sub_branches_right),results = best_branch.run()
        branches = sub_branches_left + brances + sub_branches_right
        for r in results:
            yield r
```

Listing 4.8: The dissolution algorithm in pseudo-python

In the following we will briefly describe the implementations of various combinators of `HCntT`.

Alternative/MonadPlus

The most fundamental combinator for any backtracking monad is the one splitting branches, the implementation of `Alternative` or, equvalently in our case, `MonadPlus`. The implementation for `HCntT` is fairly straightforward (listing 4.9) is fairly straightforward: for `empty` or `mzero`, which indicates failure of a branch, simply disregards the continuation and returns an empty `HRes`. The `mplus` or `<|>` simply returns an `HRes` with only the two alternative branches having maximum priority (i.e. `mempty :: HeapKey h`) and bounded to the current continuation. Both those branches are pushed from the right so that they are scheduled before other same-priority branches.

```
instance (IsHeap h,Monad m) ⇒ Alternative (HCntT h r m) where
  ContT m <|> ContT m' = ContT $ \f → return
    $ HRes (Tup2 (h (m f) <> h (m' f)) mempty,[])
    where
      h = singletonHeap mempty
  empty = ContT $ const $ return emptyHRes
```

Listing 4.9: The implementation for `Alternative` is the same as the implementation for `MonadPlus`.

halt: passing control to the scheduler

We mentioned the `halt` `HCntT` process. Because we want to be able to transform the `HCntT` monad we define the class of `MonadHalt` that can support this operation. Most common monad transformers of `HCntT` can trivially support halt. The implementation of `halt` for `HCntT` itself is demonstrated in listing 4.10. The provided priority is offset by the previous priority of the branch.

```
class Monad m ⇒ MonadHalt  m where
  type HaltKey m :: *
  halt :: HaltKey m → m ()

instance (Monad m,IsHeap h) ⇒ MonadHalt (HCntT h r m) where
  type HaltKey (HCntT h r m) = HeapKey h
  halt v = ContT $ \nxt → do
```

```
    v' ← asks (v <>)
    return $ HRes (Tup2 (singletonHeap v $ nxt ()) mempty,[])
```

Listing 4.10: The halt process yields updates the priority of the branch and yields execution to the scheduler.

Soft-cut/eitherl

The `<//>` (pronounced eitherl), `left <//> right` runs the left hand side operand (primary) and if no values are produced in the *entire* computation based on that, only then does it try to evaluate the right hand side (fallback). `HCntT` does not guarante that the right hand side will be scheduled immediately after it realizes that there are no solutions, but it does guarantee that it will be scheduled before it moves on to a new priority. In other words it is only guarantted that the priority of the fallback branch will tie the last failing branch of the left hand side in terms of priority, but if there are other branches that tie, there is no guarantee of how those will be scheduled.

There are two challenges that our particular feature set imposes to implementing this:

- We want `<//>` to operate on the entire operation, unlike [73] that makes the decision o whether to run the fallback solely based on whther the left hand side returns a value.
- In the context of a weightedx search control needs to be able to escape a branch that passes through the left hand side operand before it is exhausted. Therefore we can not simply dissolve the left hand side and reconstruct it. But then how would the operator know when the branches are exhausted?

We solve these problems with the use of a special kind of branch we call a *marker*, and a global state. Since branches are arbitrary processes we equip them with access to global state (local to the computation), a lookup table (`CompState`) full of fallback branches. The main ideas is that a fallback branch corresponds to an entry int the lookup table. This entry is modified accordingly by the children of the right hand side. At the time of the cut we push a marker into the heap with priority lower than the child branches so that when it is scheduled it will perform some checks based on the lookup table entry and schedule the fallback function if all the children branches have finished whithout a result.

We will see in a bit how the contents of the lookup table are updated and how new, lower priority markers are pushed in the heap. For now consider that each marker may be superseded by a lower priority marker rendering it a invalid. At any time there is exactly one valid marker per active fallback and it is denoted as such by the fallback entry in the lookup table. The rest of makers are invalid and should be equivalent to noops.

More precisely about the internals of the marker processes, each one refers to a location in the lookup table via its closure. Also each marker is uniquely identifiable so each valid entry in the lookup table references one marker. Specifically the lowest priority one that corresponds to it. When the marker is scheduled it looks up the fallback branch entry in the table and checks if the entry also refers to that marker. There are 3 possible scenaria that may play out at this point:

- The fallback entry in the table has been invalidated by a branch that yielded a result. In this case the marker is invalid and just returns.
- The fallback entry is valid but does not correspond to the scheduled marker. This means due to the left hand side branch spawning low priority sub-branches, another marker has been inserted to (possibly) trigger the fallback at some time in the future. The current marker is invalid
- The fallback location is valid and corresponds to the scheduled marker. This means that it is time fallback process needs to be run and removed from the table.

But what is the lifecycle of the fallback entries? When an exprssion `A </> B` appears we create a new fallback entry in the lookup table containing `B` and recursively "infect" all spawned sub-branches of `A` to perform the following actions immediately after they generate new branches and results:

- If there is at least one valid result invalidate the fallback in the lookup table and stop infecting child branches with the currently described hook.
- If the fallback is invalid it means there have already been valid results that rendered the fallback obsolete. Stop infecting sub-branches.
- If none of above happened check the priority of the last marker corresponding to the fallback (registerd in the lookup table entry). If it is strictly lower than the lowest priority subbranch do nothing because there is a well placed marker to handle it.

> Otherwise create a new marker of the same priority as the lowest priority subbranch and put it in the right-append heap. This way we know it will be scheduled AFTER the last branch relating to the fallback.

It is demonstrated in figure 4.1

There are some optimizations that can be implemented to avoid too many lookups in the fallback table in the case of deep eitherl nesting that take advantage of the fact that new markers for outer fallbacks may only be created when new markers for inner fallbacks are created. However, for FluiDB we did not find `CntT` to be too bad of a performance bottleneck so we leave it for a future iteration.

Once

The `once` operator runs the argument computation and stops once it returns the first result. FluiDB uses `once` to run the garbage collector without exploding the search space. Generally the different combinations of nodes that can be deleted is huge. We prune that by space by settling for the first result the planner can find.

The way `Once` is built on top of a concept we call a *nested scheduler*. It takse a computation (the *nested computation*) and needs to know what to do in case of a success and in case of complete failure (no more branches to run). The nested scheduler is a process that *always* returns a single case subbranch. This subbranch has the priority of the highest priority subbranch of the nested computation and when scheduled it internally schedules the next branch of the nested process. When the process yields results or fails completely the corresponding hook is run. `once` implements these hooks as "return the result and stop" and as "propagate the failure" respecively. Since the implementation of `once` is fairly small it is provided along with the types in listing 4.11.

```haskell
nested
  :: forall h m r a .
  (Monad m,IsHeap h)
  ⇒ (Tup2 (h (Brnch h r m)) → r → [r] → HRes h m r)
  → HRes h m r
  → HCntT h r m a
  → HCntT h r m a
nested success fail c = ...

```

```
once :: forall h m r a . (Monad m,IsHeap h) ⇒ HCntT h r m a → HCntT h r m a
once = nested (\_h r _rs → HRes (Tup2 mempty mempty,[r])) emptyHRes
```

Listing 4.11: The nested scheduler runs a subprocess within a single branch. Once is built on top of that to make sure the process stops once a result is returned.

It is worth noting that nested schedulers are also applicable of implementing `<//>`. In the marker-based implementation we are performing N lookups on the fields table, N being the nesting of `<//>` operators. Implementing it using nested schedulers, on the other hand, means we are doing N pop/push pairs, one for each nested heap. For FluiDB we use a slighltly too general notion of a heap so we opt for the former.

4.2 The planner

4.2.1 Traversing the graph

The planner is the subsystem of FluiDB that given the state of the QDAG and a target node to be materialized produces a logical plan that will materialize the query. Our notion of a logical plan is slightly more specific them what is commonly considered a logical plan, i.e. the RA representation of the query. In our case it is a sequence of *transitions* that are to be transpiled to C++ by the code generator. There are three kinds of transitionse:

- t-node trigger that assumes the input n-nodes are materialized and produces a subset of the output nodes.
- t-node reverse trigger that assumes that the output nodes are materialized
- n-node deletion

The planner operates by backtracking using `CntT` monad. Each branch mainains some branch-internal effects that are reified as monad transformers on top of the `CntT` defining the `PlanT` monad transformer (listing 4.12).

```
type PlanT t n m =
  StateT
    (GCState t n)
    (ReaderT (GCConfig t n)
     (ExceptT (PlanningError t n)
      (HCntT PlanHeap () m)))
```

Listing 4.12: The monad that defines all the useful effects used by the planner. `GCConf t n` is an immutable, from the persepctive of the planner, configuration that includes the QDAG, the node sizes, etc. `GCState t n` is state that is mutated and private to each branch of the planner like the materialized status of the nodes, the set of transitions registered so far and various caches. The `PlanningError t n` is a planner specific type of error. The entirety of the result of planning is accumumated in `GCState` so the result of backtracking is just unit (`()`).

It is important to clarify the way we use the term "materialized node" (`Mat` as opposed to "not materialized" – `NoMat`) from the planner's perspective. A mapping of node states is passed to the planner at the beginning of the planning process. This mapping indicates which nodes are initially materialized. Then as transitions get registered update the mapping is updated to reflect the effect that the plan so far would have on the materialized relation set in storage. Since the planner operates via backtracking (using the `CntT` monad described), each branch maintains its own mapping of node states and therefore considers a different set of materialized nodes.

We maintain the partial plan (sequence of transitions) as part of the state of each of the planners branches. Registering a transition means that a transition is added to the partial plan.

The main loop of the planer is a recursive process of asseerting that nodes are materialized. If the node is already materialized the assertion succeeds, if not the planner attempts to trigger, or reverse trigger, a neighbouring t-node that would lead to the node being materialized, asserting first that the input nodes of that transition are materialized. In practice, due to intermediate n-nodes being auxiliary and not corresponding to real nodes, we know that t-nodes are organized in sequences that need to be triggered entirely or not at all. For example (figure 4.2) the intermediate $\ltimes$ nodes are not actually materialized as part of the join operator. Equipped with this knowledge we organize the t-nodes into `MetaOps` (listing 4.13) that can atomically be triggered or reverse triggered. `MetaOps` also abstract the distinction between triggering and reverse triggering as they expose a set of input, a set of output nodes and a process that registers the correct transitions once the `MetaOp` is triggered. The process of asserting asserting an n-node being materialized then becaomes the process of non-deterministically selecting a `MetaOp` with the node in question in the output set. Before actually splicing the `MetaOp` compiutation we assert that the input set is materialized.

```haskell
data MetaOp t n = MetaOp {
  metaOpIn     :: NodeSet n,
  metaOpOut    :: NodeSet n,
  metaOpInterm :: NodeSet n,
  metaOpPlan   :: forall m' . Monad m' ⇒ PlanT t n m' [Transition t n]
  }
```

Listing 4.13: A `MetaOp` refers to input, output, and intermediate nodes that are involved in the set of operations it abstracts. Furthermore it contains a computation that registers and returns the transitions involved in the `MetaOp`.

Two questions should arise from the above description: a) how does the planner avoid cycles where it recursively tries to materialize the parents and then the children? and b) how does it know not to materialize a node twice? Both those problems are addressed by refining the possible states of the n-nodes (listing 4.14). Rather than being just `Mat` or `NoMat`. We also disambiguate between `Initial` and `Concrete` nodes. Nodes in the `Initial` state are allowed to change their materialization status. In contrast `Concrete` nodes have their state fixed with very few exceptions. Then, when a node is asserted to be materialized its status is first checked and followin scenaria are possible:

- the node's status found to be `Concrete` and `NoMat` and the branch immediately fails as it is not allowed to be set to materialized.
- if it is `Concrete` and `Mat` the assertion simply succeeds
- If it is `Initial` and `Mat` it is turned into `Concrete` and `Mat` and the assertion succeeds.
- if the node's status is `Initial` and also `NoMat`, two things need to happen: first the node is set `Concrete` and `NoMat` in order to avoid cycles, and the aforementioned process of finding a `MetaOp` is carried out recusively in order to materialize the node.

```haskell
data IsMat = Mat | NoMat
data NodeState =
  -- Concretified states are allowed to change only within the same
  -- lexical scope.
  Concrete IsMat
  -- Initial states are subject to change when encountered as a
  -- neighbour or by the GC.
  | Initial IsMat
```

Listing 4.14: The different states that a node is allowed to be in.

Once an n-node's is materialized and we start materializing its siblings, it is important that the garbage collector (which we will be discussing in the next section) does not delete said n-node. To avoid this problem, items materialized are set to `Concrete Mat` until all their siblings, that are inputs to the same `MetaOp`, are materialized. Once the `MetaOp` is triggered and its output nodes are set to the `Concrete Mat` state the input nodes are set to `Initial Mat` as it is now safe for the garbage collector to remove them.

It is worth noting here the importance of every node in the depset being completeley materialized before the process of materializing the next one comences. The reason is that at any time a single trail of parent nodes is marked as `Concrete` and `NoMat`, otherwise we can't be sure whether a `Concrete NoMat` node that renders a `MetaOp` non-triggerable is actually in the process of becoming materialized, meaning that when that process succeeds the `MataOp` under consideration will be triggerable.

4.2.2 Garbage collection

One of the fundamental design decisions of FluiDB is that it is commited to materializing all intermediate nodes and keeping them around for as long as possible. In one hand the plan selected is crafted so that the intermediate results are maximally useful for the overall workload, on the other, when the available storage runs out, the garbage collector mechanism selects the least useful nodes to be deleted, creating free required for solving the query. In this section we focus on the latter.

The garbage collector is triggered right before a `MetaOp` is to be triggered. It infers the space required for materializing the output nodes and the space available to decide how much space is required. Then it selects a subset of materialized nodes to delete in order to make room for the new results. The selection process has two hard constraints:

- The nodes being deleted must be *deletable*, i.e. the node's state is `Initial` and if the node is deleted it can be reconstructed from the remaining materialized nodes.

- The nodes being deleted must not be required for the continuation of the current plan. We call these nodes *protected*. For example, say we are planning for the query $A \bowtie B$ and we have materialized A already but while materializing B the GC is triggered. A should not be in the repertoire of nodes the GC can delete under any circumstances.

Selecting a subset of nodes is not an easy problem so we follow a simple heuristic and leave a proper solution for future work.

Large nodes are costly to create and therefore the larger the node, the less inclined the planner is to create it, and the more useful it is likely to be. With that in mind the heuristic we follow is for the garbage collector to prioritise deleting small nodes over deleting larger ones. The first order of business for the GC the is to find the set of deletable nodes and sort them by size. It tries to delete them one by one starting from the smaller ones and working its way up to the larger ones. Every time a node is deleted it is possible that other nodes that were previously established to be deletable lose that property. For example if all nodes A, $\sigma_p A$, and $\sigma_{\neg p} A$ are **`Initial`** and materialized, each one of them can be materialized from the others and therefore all of them are marked as deletable. If the GC deletes $\sigma_p A$ first the other two are no longer materializable as they depended on A to be materializable.

Nodes that are established as non-deletable in this way are switched from **`Initial`** to **`Concrete`** to avoid re-calculating their materializability.

If after this process not enouch space was created, the GC resorts to starting a new *planner epoch*. When a new planner epoch begins all the nodes states and transitions that were recorded since the last planner epoch started are stashed into the epoch stack. The epoch stack resides in the branch-local state (**`GCState`**) and contains a map of nodes to their states at the end of the epoch and a sequence of transitions. When a new epoch is pushed into the stack a new mapping of states is created according to the following rules:

- The materialization status does not change between epochs: All materialized nodes from the previous epoch are materialized in the new one materialized and all non-materialized nodes are still not materialized.

- All non-protected **`Concrete`** nodes become **`Initial`** nodes.

The sequence of transitions for the new epoch is empty. The reason we keep separate lists of transitions is, as we will see in more detail in the code generation chapter, that the subset of output nodes that a physical operation generates is established at the time of physical planning based on the materialized nodes according to the epoch corresponding to the transition.

An epoch contains all the important information about the state of the planner. For this reason when an epoch is inserted in the stack it is checked for equality against the other epochs in the stack. If two equivalent epochs are inserted in the stack the branch fails permanently as it means that there is a cycle.

This enables another optimization, the *free-to-delete* nodes. As mentioned previously the planner prioritizes the version of **MetaOps** that materializes all the outputs for example it will prioritize the branch that followes $MetaOp\{in = \langle A\rangle, out = \langle \sigma_p A, \sigma_{\neg p} A\}$ over the on that follows $MetaOp\{in = \langle A\rangle, out = \langle \sigma_p A\}$. It is rarely the case, however, that all the outputs are required. In the example just mentioned then the former example, triggering the former **MetaOp** and then garbage collecting $\sigma_{\neg p} A$ without it being used is never desirable. For that reason we mark the node $\sigma_{\neg p} A$ as *free-to-delete* as soon as it is created based on the fact that it is not required. When the garbage collector deletes free-to-delete nodes it does so without registering a deletion transition. This way the physical planner will generate a plan where $\sigma_{\neg p} A$ is never created in the first place. All nodes lose their free-to-delete status upon the creation of a new epoch.

The comprehensive algorithm for the GC is presented in listing 4.15

```
gc reqSize = do
  -- Try the current epoch and if that fails retry with a new epoch.
  return () <//> newEpoch
  -- find the deletable nodes and sort them by size
  deleteables ← sortOnM getNodeSize =<< filterM isDeletable =<< getAllNodes
  -- Try deleting each node and stop deleting when amassing enough
  -- free pages.
  forM_ deletables $ \n → do
    freePgs ← getFreePages
    when (freePgs < reqSize) $ tryDelete n <//> markAsConcrete n

tryDelete node = do
  guardM isDeletable
  isFreeToDel ← getIsFreeToDel node
  unless isFreeToDel $ register $ DelNode node
  setStatus node (Initial NoMat)

isDeletable n = do
  st ← getNodeState n
  case st of
    Initial Mat → do
      setNodeState n (Initial NoMat)
      ret ← isMaterializable n
      setNodeState n (Initial Mat)
      return ret
    _ → return False

markAsConcrete = do
  st ← getNodeState n
  case st of
    Initial m → setNodeState (Concrete m)
    _ → return ()
```

Listing 4.15: A sketch of the garbage collector algorithm in pseudo-haskell. The `<//>` operator is supported by the **`HCntT`**.

The garbage collector may have a final trick up their sleeve when all else fails: neighbor materialization. When there exists a dependency set of a non-deletable node that takes up less size that the node itself, the transition, the GC can attempt to trigger the corresponding **`MetaOp`** in order to generate the dependency set and render the node deletable. For example the θ-join node $A \bowtie_\theta B$ is likely to take up more space than A and B combined. This makes the search space explode and is a generally low-yield strategy, therefore it is by default disabled.

It is hopefully clear at this point that the process of garbage collection involves a huge search space. We sacrifice a some of our plan repertoire by running the entire process wrapped in a `once` operator that we described in the previous section.

Finally, a word about protected nodes. Nodes are protected within the context of a branch from the time they are established as part of the input set of a **`MetaOp`** we are making trigerable until the time said **`MetaOp`** is actually triggered during the normal planning (i.e. not the GC). Because a node may be the input of more than one simultaneously considered **`MetaOps`**, protection of a node is not a boolean value that is set when the node is encountered as **`MetaOp`** input and unset when the **`MetaOp`** is triggered, but rather an natural number variable that is incremented and decremented respectively. When that value is zero, the node is not considered to be protected.

4.2.3 Order of traversal

We mentioned that the **`MetaOps`** are selected non-deterministically. In this section we will go in depth on the FluiDB planner's strategy on the order in which it considers its options using the **`CntT`** framework, making heavy use of the `halt` operator to set priorities for branches. A branch's priority is dependent on the particular frontier at the time of a halt and is determined by four factors:

- The cost of the MetaOps that the branch is in the process of making trigerable.
- The cost of the MetaOps that have already been triggered.
- The sum of the estimated costs of each node in the frontier.
- A weighted sum of the *stochstic cost* of historical queries based on the current frontier.

The final two factors are values, on which we will focus in this section, are computed often and are dependent on the set of materialized nodes. Since the set of materialzied nodes does not change in a completely arbitrary way from computation to computation a naive approach to calculating those values would involve a lot of duplicate work. For this reason we use Antisthenis which will be expanded in a separate chapter to minimize the amount of work required.

Let's start with the sum of estimated costs. We have a very rough way of estimating costs. We incrementally estimate the cost of each node of the frontier using the following formula:

$$c_{nomat}(n) = \min_{op \in \text{metaops with } n \text{ output}} \left\{ cost(op) + \sum_{i \in inputs(op)} c(i) \right\} \tag{4.1}$$

$$c_{mat}(n) = 0 \tag{4.2}$$

Where the cost c is c_{mat} if the cost is materialized and c_{nomat} otherwise. In plain english the cost is recursively estimated as cheapest combination `MataOp` plus the cost of materializing the input nodes.

This approach has several problems like the fact that nodes that are used more than once are added more than once and that it does not take at all into account the budget constraints. It is, however, a good enough heuristic for prioritizing the branch.

The final factor, which takes into account the historical queries is a bit more tricky. The planner keeps track of the last couple of nodes materialized previously and tries to estimate how useful beneficial following the particular branch would be in the event where a query similar to those is requested in the future. We estimate that by summing up a notion of cost for those queries.

A naive approach would be to just use the the same algorith as we did for the frontier nodes. However, this approach is prone to getting stuck behind materialized nodes. However, this approach is prone to getting stuck behind materialized nodes. Past queries, epsecially recent ones are likely to still be materialized and therefore their cost is likely to be 0. Even if the GC has gotten around to deleting them, their near dependencies are likely to block the cost estimator go far. This makes the estimation quite bad for two reasons:

- The planner is likely to have deleted any number of materialized nodes that the cost estimation may be depending on.
- We don't care about the cost of the particular nodes, but rather about *similar* nodes. For that reason we want to avoid depending too heavily on a particular materialized node being there.

A realistic way of calculating the expected cost of a node in the future, which we very informally and heuristically attempt to approximate, would be to instead of considering the cost of materialized nodes to be zero, to calculate the likelihood that a materialized node will still be materialized when we encounter it again. This is related not only to an estimation of how many page-writes separate the moment of cost estimation and the actual plan the cost of which is being estimated, but also all the decisions that the garbage collector will make in that time. After FluiDB's budget is exhausted for the first time, for every page write there a page needs to be garbage collected. We considered a couple of options for stochastically modelling the behavior of the GC like assuming that it chooses random pages or random tables, but we could find none that was useful and computationally viable.

For this reason we decided to follow a pragmatic approach to the problem and simply assume that for every materialized node there is a constant probability that it will not still be materialized when we need it, scaling the cost by that factor to get the stochastic cost. So the cost formula h for the nodes now is:

$$\begin{aligned} h_{nomat}(n) &= \min_{op \in \text{metaops with } n \text{ output}} \left\{ h(op) + \sum_{i \in inputs(op)} cost(i) \right\} \\ h_{mat}(n) &= \lambda \cdot h_{nomat}(n) \end{aligned}$$

Where $\lambda \in (0,1)$ is the estimated probability that the the node is still materialized when needed. As before materialized nodes have cost $h_m at$ and non-materialized nodes have cos h_{nomat}

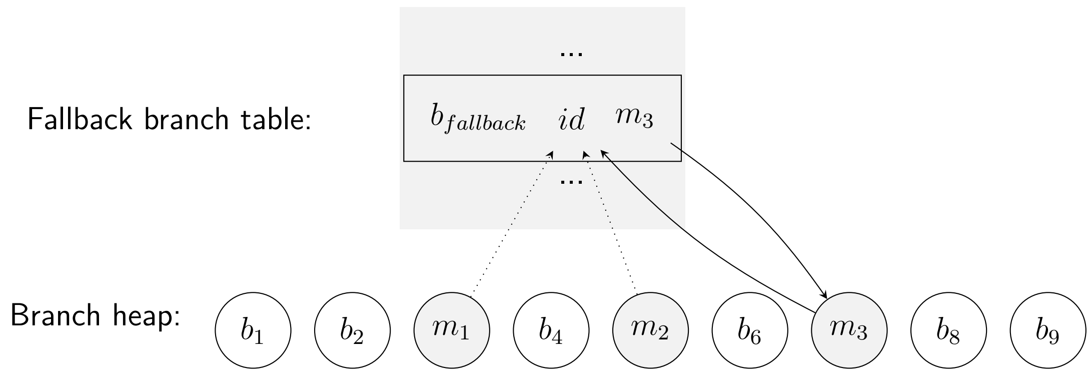

...
Fallback branch table:
$b_{fallback}$ id m_3
...
Branch heap:
b_1 b_2 m_1 b_4 m_2 b_6 m_3 b_8 b_9

Figure 4.1: Marker "branches" are injected in the heap of prioritized branches to possibly trigger the fallback branch. Each marker gives the scheduler the opportunity to run a particular fallback branch. The entry of the fallback branch entry references the last marker so that only that one actually trigger the fallback branch. A child branch that succeeds removres the fallback entry so the final marker also fails to tigger the fallback.

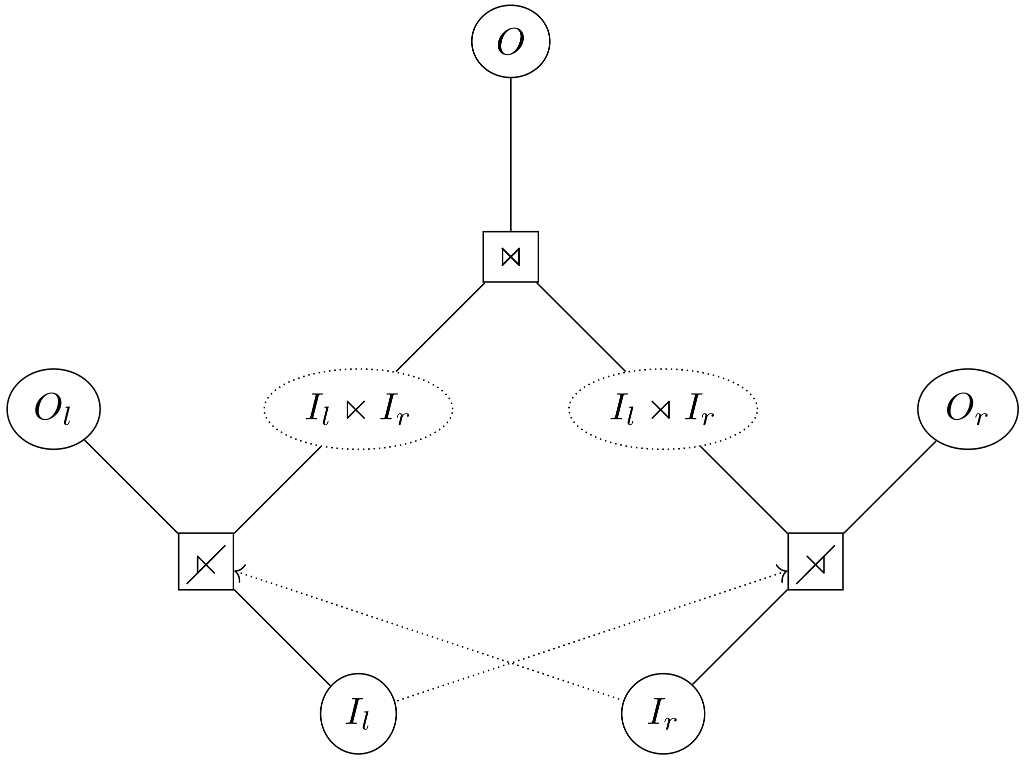


Figure 4.2: starting at node O, the output of a join operation, we can derive three MetaOps that can materialize it. $MetaOp\{in : \langle I_l, I_r\rangle, out : \langle O\rangle\}$, $MetaOp\{in : \langle I_l, I_r\rangle, out : \langle O_l, O\rangle\}$, $MetaOp\{in : \langle I_l, I_r\rangle, out : \langle O, O_r\rangle\}$, $MetaOp\{in : \langle I_1, I_2\rangle, out : \langle O_l, O, O_r\rangle\}$. Because trying all combinations of outputs explodes the seach space we always go for the largest and then let the garbage collector deal with the possible reprecussions. On the other hand to materialize I_l there is only one `MetaOp` that relates to this cluster $MetaOp\{in : \langle O, O_l\rangle, out : \langle I_l\rangle, interm : \langle I_l \ltimes I_r\rangle\}$

CHAPTER 5

Antisthenis

There is no horse, there is horses

(Antisthenis)

Chapter summary

- *Antisthenis* is a system for incremental evaluation of algebraic expressions using heuristics to optimize the order in which subexpressions are evaluated and to calculate some classes of self referrential computations.
- Mealy arrows are a construct to implement adaptable computations that can can prodice a value and a new, semantically equivalent computation. Each iteration adapts to the previous caching values and changing the evaluation plan to be more efficient.
- Antisthenis computations are implemented as network of mealy arrows that compose larger adaptive (incremental and partial) computations.
- Machines are named processes that can be referred to by other machines. Antisthenis can handle some classes of self referrential computations.
- We implement antisthenis operations to cover requirements of the FluiDB physical planner:
  - calculate min/sum of natural numbers – used to compute cost
  - boolean functions – used determine node materializability
  - calculate min/sum of numbers annotated with certainty metrics – used to calculate the cost of materialization of nodes give nodes that *might* be magterialized.

This chapter goes in depth descibing the various parts that compose Antisthenis, a system we developed to assist the FluiDB planner in incrementally and efficiently computing costs and checking if nodes in the QDAG are materializable during the evolution of the inventory.

The chapter starts with an introduction about the motivation for developing Antisthenis as a collection of synchronously communicating processes performing incremental computation. Next we look at some background concepts most of which relate to notions for computation and our conception of a Mealy arrow that builds on those notions to enable the construction of Antisthenis processes. In the following section we describe the core of Antisthenis, which is the internal construction of a single antisthenis process. From there we move on to describe some specific operation implementations that are useful for FluiDB and then describe the systems that facilitate intercommunication and high level organization of Antisthenis processes. Finally, we conclude with some shortcomings and some future future work that can be done to make Antisthenis easier to apply to solutions other than FluiDB.

5.1 Introduction

The order in which an expression is evaluated can be important not only for the performance of the expression evaluation but, in the context of infinite expressions, for the very termination of the evalution itself.

Expressions are typically modelled as abstract syntax trees or computation graphs, and evaluation is commonly modelled as a reduction of that tree or DAG. Depending on the evaluation strategy each operand is either fully or partially evaluated. Typically, the strategy for evaluating the subexpression operand however is oblivious to the parent expression and a partially evaluated subexpression (thunk) is opaque from the perspective of the parent expression. Breaking this assumption provides some opportunities for incremental computation and efficient evalaution in the presence of non-total arguments.

We built *Antisthenis* around this principle to achieve three important goals:

- Incrementally evaluate an expression, i.e. to only re-evaluate the parts of it that depend on parameters that were updated with respect to the last time the expression was evaluated.
- Take advantage of absorbing elements and other operator-specific strategies for

early stopping the evaluation.

- Often sub-expressions are not total, i.e. they may not be computable. A common reason for that is the expression being self-referential. These errors may be detrimental to the evaluation the top level expression, but often as we will see they are not.

Incremental computation is a tool for efficiently evaluating expressions of variables that change over time. The area has received some attention recently [32, 42]. An incremental evaluator will typically accept a directed acyclic graph (DAG) of interdependent computations like the following one

$$\begin{aligned}A &= a + B + C + D\\B &= C \times b\\C &= D + c\\D &= 0\end{aligned}$$

Which can be represented as a dag in figure 5.1.

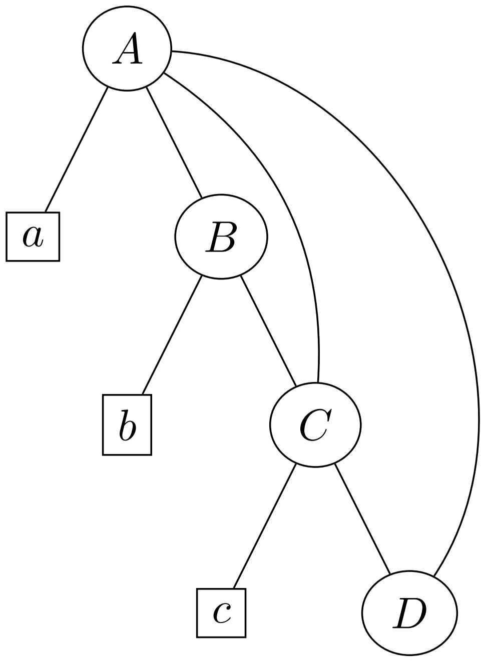


Figure 5.1: A computation DAG.

Where A, B, C, and D are expression nodes in the DAG and a, b, and c are parameters that change over time. A depends on all three parameters. A non-incremental evaluation system would evaluate all four nodes every time a value were requested, regardless of whether and which parameters have changed. Incremental evalutaion systems keep track of which variables change and only re-evaluate the nodes of the DAG that have been updated. In our example if only the parameter b has been updated since the last time A was evaluated, in the meantime only A and B would need to be re-evaluated.

Incremental evaluation systems employ a wide range of tricks from simply memoizing the functions [5] to more complex strategies like [42] where changes in parameters are propagated in the form of dirtying DAG nodes and more explicitly breaking the computation into reusable parts.

To our knowledge none of the existing approaches to incremental computation take into account the presence of either absorbing elements or non-compuatable subtrees (i.e. recursive values) in their scheduling the evaluation order. It is either done naively or left up to the programmer. However, it is a hard requirement of the FluiDB query planner for the incremental computing framework to automatically exploit the properties of the computation at hand in order to incrementally compute node materializability and query costs in the presence of different materialized query sets. Furthermore, in all the approaches we are aware of the network of sub-expressions was assumed to be well-behaved, in that all nodes return a valid value and there are no cyclical references.

The main principle behind the architecture of Antisthenis is to manipulate the order in which sub-expressions are evaluated in order to exploit properties of the parent operator to avoid fully evaluating each of the sub-terms. For example when evaluating the cost of a database query we take the result of the computation is the minimum cost of each of the possible logical plans. It is unlikely that one would need to come up with a full estimation of each plan, especially the more expensive ones, before realising the lowest cost. Instead, we would like to accumulate a lower bound for the cost of each plan.

Another prime example of an opportunity pruning the expression tree is the exploitation of absorbing elements. In the general case, in a non-lazy evaluation strategy, to compute the value of a function $f(A, B, C)$ where A, B and C are expressions, we would need to fully evaluate *all* three arguments first. Since the absorbing element of multiplication in the reals is 0, for $f(a, b, c) := a \times b \times c$, a result of 0 for the first argument A would render the evaluation of the rest of the arguments futile.

As alluded to, a lazy evaluation strategy paired with a Sufficiently Advanced Compiler, can take steps towards mitigating this problem. This can only go so far, however. Consider the following example:

$$
\begin{aligned}
A &= B \times C \times D \\
B &= \sum_i i \\
C &= 0 \\
D &= \sum_i i
\end{aligned}
$$

Here, when evaluating A, a traditional evaluation system would look at the first argument first, namely B. B is very expensive to evaluate and very soon a human would figure out that it will never evaluate to zero. C, however, does evaluate to zero, and once that is evaluated the evaluation of A is complete. In a complex set of automatically generated such expressions it is important that the evaluator does not fall into a deep expression when a very shallow one could yield an absorbing value.

Early stopping a computation is not useful only to avoid extra work. As we mentioned, many incremental computation systems assume that all sub-computations terminate successfully. Due to the presence of cycles in FluiDB graphs we need to take seriously the case where a sub-expression will fail to terminate. In this case the parent expression needs to exhaust all possibilities that could lead to a value.

$$
\begin{aligned}
A &= min(B, C, D) \\
B &= b_1 + b_2 \cdot D \\
C &= c_1 + c_2 \cdot A \\
D &= d_1 + d_2 \cdot B \\
b_1 &= b_2 = d_1 = d_2 = 1 \\
c_1 &= 3 \\
c_2 &= 0
\end{aligned}
$$

Here B and D are clearly not computable but in the context where these values represent cost of evaluations for nodes A should evaluate to 3 and A should evaluate to

$C = 3$. Furthermore, A should depend *all* parameters for the validity of its value while C should depend only on c_1 and c_2.

5.2 Background

In order to describe our model of self-adjusting computation we need to lay the groundwork by describing the basic model of computation as understood by ML like languages, and specifically Haskell. These languages define computations as first class values that have parametric types. A computation of type `f a` can be understood as a computation that when executed would generate a value of type `a`. In that sense `f` is a generic type with one type parameter `a`, thereby being of kind $\star \to \star$, whereas `a` and `f a` are both of kind $\star$. For example, in Haskell, we represent as `IO Int` the type of a computation that interacts with the outside world in order to compute an integer, e.g. by reading user input, by writing files, or, as the meme in the community goes, by launching missiles.

From the perspective of a programming language in order to manipulate computation it is useful to be able to change computations, i.e. to be able to create morphisms `f a → f b` from simpler primitives in the language. The notation of a the arrow `→` used here is a function within the basic framework of the language, a simple mapping between a value of type `f a` to a value of type `f b`. This is indeed already computation carried out by the runtime. In that sense it is *implicit* computation. We have little control over the conditions under which it is carried out, especially in a lazy language like haskell, and we consider it to be pure, with no effects. This will become important when we discuss arrows and profunctors.

Returning to our higher kinded (of kind $\star \to \star$), explicit computations, commonly in haskell a hierarchy of 3 typeclasses (interfaces) that computations conform to has been established, each of which provides a different way of producing morphisms of the computation itself, and therefore each of which allowing different computations to be expressed by type `f`. While these typeclasses are more general than refering only to computations, we will try to provide an intuition of each for them in reference to computations, to avoid overwhelming the reader with the full generality of these constructs. The reader is encouraged to notice how the weaker, more general, typeclasses presented first, more intuitively describe containers of values and as we strengthen the constraints and require more operations of our objects it becomes harder to think of containers that fit the constraints and notions of the values as computations becomes more natural.

5.2.1 Functor

For computations implementing the `Functor` typeclass we can create a morphism of the computation from a normal haskell function. For every functor `f` then, there must be a function `fmap :: (a → b) → f a → f b` that abides by the functor (see listing 5.1) [22]. In plain english, given a computation and a function that can transform the result of the computation we can get a new computation that is equivalent to the original computation but yielding the transformed result. The functor laws (see listing 5.2), which all valid implementations of a functor must follow, assert that applying the identity function to the result of a computation does not change the computation itself and that `fmap` ing has no other effect on the computation other than changing its result.

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

Listing 5.1: The functor inteface in haskell.

It is common to understand functors in the context of haskell as mappable containers that can hold any type value. One could think of a computation that supports the functor interface as a computation that results in a functorial container. An example of a computation that is *not* a functor might be one that produces a set of values, because a set of values requires that the values are comparable, and not all per element value transformations of a set are valid. For example a set of integers `Set Int` can be unambiguous for the language but a set of functions `Set (Int → String)` is not unless we define a notion of equality for functions.

```haskell
-- Identity
fmap id == id
-- Composition
fmap (f . g) == fmap f . fmap g
```

Listing 5.2: Laws that any valud functor inteface interface must obay.

5.2.2 Applicative

Computations that are applicative [22] functors are computations that can be combined in "no particular order". In particular an applicative functor must implement `ap :: f (a`

`→ b) → f a → f b` (more commonly written as `<*>` which is a bit harder to pronounce) and `pure :: a → f a` (see listing 5.3). Applicatives are more often described in terms of computation than functors. Firstly, this means that we must be able to construct a trivial computation that just returns a given value, meaning that there must be no restriction on the kinds of values an applicative computation can yield. For example a computation that yields only integers is not an applicative because then there would no way to universally quantify the argument of `pure`. Incidentally it is not a functor either for the same reason, namely that we wouldn't be able to universally quantify the output of the input function of `fmap`.

```haskell
class Functor f ⇒ Applicative f  where
  (<*>) :: f (a → b) → f a → f b
  pure :: a → f a
```

Listing 5.3: The interface of a haskell applicative functor.

Furthermore, from the definition of `ap` or `<*>` we understand that two computations that are applicative functors can be combined into one computation with no restriction on the order thtat they are evaluated. There are a few examples of applicatives being used to represent parallelizable computation, the most prominent of which being Facebook's Haxl [39]

Like with `Functor` the interface of applicative must be subject to the applicative laws presented in 5.4. Essentially these formalize the triviality of `pure`.

```haskell
-- Identity
pure id <*> v = v
-- Composition
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
-- Homomorphism
pure f <*> pure x = pure (f x)
-- Interchange
u <*> pure y = pure ($ y) <*> u
```

Listing 5.4: Laws that any valid applicative intreface must obay

5.2.3 Monad

A meme in the functional community defines monads as "monoids in the category of endofunctors". To the unfamiliar reader this may take a second to parse, but it is really just a fancy way of saying something fairly simple: we can transform any nested monadic functor `f (f a)` into a flat one `f a`, and if the stack is large `f (f ( ... f (f a)))` the order in which the functors are collapsed does not matter. In the context of computations this implies that nested computations run from inside out, the inner computation is run first and then the outer one is run. The "and then" part is what a monad is meant to bring to the table. Viewed as computations monads must have all the properties of functors and applicatives but they must also implement the operation `(>>=) :: f a → (a → f b) → f b` where `>>=` is pronounced `bind`. The computation described by the first argument of `>>=` must be executed **first** in order to generate the argument for the function in the second argument which must **then** produce the result. We commonly represent a monadic functor by the characer `m` rather than the character `f` that we used for functors and applicatives. Monads are the most commonly used abstraction to describe computations comprised of interdependent steps. The haskell interface for monads is succinctly presented in listing 5.5

```
class Applicative m ⇒ Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

Listing 5.5: Definition of the interface of a haskell monad.

Monad implementations need to adhare to laws laid out in listing 5.6. These laws are similar to the applicative laws in that they formalize the trivality of `pure`, now called `return`, in relation to the `bind` operator this time. However, unlike the applicative laws, monadic laws introduce the law of associativity that may seem strange to a reader unfamiliar with Kleisli arrows [20]. The essence of the monad, when focusing on the second argument of bind (the one typed `a → m b`), and which is codified by the monad laws is that a monad `m` must give rise to a category, referred to in the literature as the Kl category.

A category needs a set of objects and a domain of arrows that can be composed. Further we need the arrow composition to be associative and an identity arrow that maps objects to themselves. An obvious category is formed by all objects in haskell, haskell functions as arrows and `id :: a -> a` as the identity arrow. This is commonly referred to as the hask category. As aluded to earlier, this category does describe computation albeit in a more implicit manner. The monad laws assert that each monad gives rise to a kleisly category where objects are the objects of hask, arrows `a \textasciitilde {}> b == a -> m b` and the identity arrow is `return`. Moving forward we will see how, capitalizing on this conception we can come up with a more flexible/

```haskell
-- Left identity
return a >>= k = k a
-- Right identity
m >>= return = m
-- Associativity
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Listing 5.6: Laws that any valid monad implementation must abide [26].

Monad examples

To make the concept tangible we provide a few examples of how monads encapsulate computational effects by looking at four monads: `Maybe`, `State` , `IO`, and `Free`.

Maybe The `Maybe` functor (defined in listing 5.7) implments partiality. In other languages the same concept may be called `opt` or `Option`. A partial function is denoted `a -> Maybe b` because it may return a value or not return a value.

```haskell
data Maybe a = Just a | Nothing
```

Listing 5.7: Definition of the `Maybe` monad.

Composability of the Kleisli arrow allows us to compose multiple partial functions into new ones, thus creating computations that can trivially fail. `Maybe` provides a good opportunity to distinguish between `Applicative` and `Monad`. Considering the two programs in listing 5.8 where we only care about `Maybe` as an `Applicative`. The semantics of the program do not specify whether calling `invAdd` will cause `inverse a` or `inverse b` to be computed first or if they will be computed in parallel. No order is imposed between them. However, in the program presented in 5.9 when evaluating the function `f` the expression `inverse a` **must** be evaluated before the `toInt` so that `toInt` has an input to operate on.

```haskell
inverse :: Double → Maybe Double
inverse a = if a == 0 then Nothing else Just (1 / a)

invAdd :: Double → Double → Maybe Double
invAdd a b = (+) <$> inverse a <*> inverse b
```

Listing 5.8: Example usage of the `Maybe` applicative functor.

```haskell
inverse :: Double → Maybe Double
inverse a = if a == 0 then Nothing else Just (1 / a)

toInt :: Double → Maybe Int
toInt a =
  if a == fromInteger (round a)
    then Just (round a)
    else Nothing

f :: Double → Maybe Int
f a = inverse a >>= toInt
```

Listing 5.9: Example usage of the `Maybe` monad functor.

State A `State` monad (listing 5.10) encapsulates a computation that depends on mutable state of type `s`.

```haskell
newtype State s a = State (s → (a,s))
```

Listing 5.10: The state monad describes mutable state.

`State` is essentially a function that accepts some state, modifies it and returns it along with a value. In C-like languages the operator to compose different components that operate on the and local state is `;`. `f() ; g() ;` means that `f()` can have arbitrary side effects on the global state and must be computed entirely before `g()`. In our slightly more precise flavor of effectful computations `f >> g` as a `State` composes a state functor that expects some state value, passes it to `f` which modifies it and passes it to `g`.

IO The `IO a` monad is "magical" in the sense that it is completely opaque to the runtime. It represents the interaction of our haskell program with the outside world, and since all haskell functions are pure haskell can be thought of as metalanguage that defines programs composed of different `IO a` components called the `main :: IO ()`. The components of an `IO` computation are guaranteed to be evaluated one after the other.

This is demonstrated in 5.11. Because monads are so ubiquitous haskell provides some syntactic sugar called the `do` notation. The same program is presented in a much more readable for in 5.12.

The functions `putStrLn :: String → IO ()` takes a string and returns a computation that would show the string on stdout. `getLine :: IO String` is a computation that reads a line from stdin and retains its value. The `main :: IO ()` is a special variable that a haskell program needs to define, all the haskell runtime essentially does is run the IO computation that `main` refers to.

```haskell
main :: IO ()
main = putStrLn "What's your name?"
       >> getLine
       >>= (\name → putStrLn ("Hello " ++ name))
```

Listing 5.11: Sequanecing IO interactions using the `IO` monad.

```haskell
main :: IO ()
main = do
  putStrLn "What's your name?"
  name ← getLine
  putStrLn ("Hello " ++ name)
```

Listing 5.12: Sequanecing IO interactions using the `IO` monad also using the `do` notation.

Free monad The free monad is slightly more convoluted than the ones we talked about so far. Without getting to deep into what free structures are in generalk ([69] is a recommended source), a free monad is a structure that only supports the monad interface in a law-abiding way. The only difference is that instead of focusing on the >>= operator that we saw so far, it instead has a constructor corresponding to the equivalent operator `join` presented in listing 5.13.

```haskell
join :: Monad m ⇒ m (m a) → m a
join m = m ≫= id

-- and also
(≫=) :: Monad m ⇒ m a → (a → m b) → m b
m ≫= f = join (fmap f m)
```

Listing 5.13: The bind (≫=) and join operations on a monad are equivalent given that monads are also functors.

A free monad `Free f m` then is equivalent to the *functor* `f` that is stacked on top of itself zero or more times `f (f ( … f (f a)))`. It essentially turns a functor into a monad for "free" – as long as you don't mind that it actually a stack of nested functor rather than a single layer like monads are. Listing 5.14 presents a simple implementation, although the implemetation in 5.15 is more common. Free monads are particularly important for our implementation of antisthenis.

```haskell
data Free f a
  = Pure a
  | Free (f (Free f a))
```

Listing 5.14: A simple implementation of the free monad type.

```haskell
newtype FreeF f a x = Pure a | Free (f x)
data FreeT f m a = m (FreeF f a (FreeT f m a))
```

Listing 5.15: A simple implementation of the free monad type.

5.2.4 Arrows and profunctors

So far we constructed our objects depending on an underlying category (Hask) provided by the programming language to get a basic notion of arrows that represent pure functions. This is a very opaque and inflexible kind of computation but a strong base on which to build ones better suited to our needs. In this subsection we will look at a generalization of this : arrows and profunctors.

While describing the interfaces the various kinds of functors we already saw the following sub structures appearing in the types:

- `a → b` the normal pure function type. We take this for granted as it is provided and managed by the language, in our case Haskell.
- `f a → f b` A functor morphism. The standard functor stack we saw so far focused on producing such morphisms from simpler arrows.
- `f (a → b)` is known as the Cayley morphism
- `a → f b` which for an `f` being a monad is the now familiar Kleisli arrow.

We would like to generalize this notion to an parametric object that we will denote as $\rightsquigarrow$. We follow the arrow definition from Hughes [17] which dictates that an arrow must be a **category** (i.e. arrows must compose in a commuting way and there must be an identity arrow) and must also satisfy the following:

- it must commute with tuple (ie `first` and `second`).
- The Hask category must be embeddable in the category defined by the arrows (i.e. via the `arr` function)

In more concrete terms we depend on the haskell typeclasses presented in listing 5.16. In plain english an arrow can act on the first or second element of a tuple, and an `ArrowChoice` can act on the the left or the right case of an `Either`.

A related concept to arrows is the profunctor. Profunctors are not necessarily directly composable like arrows nor to they commute with tuple or `either`. They only provide `dimap`. `dimap` is of course implementable in terms of arrow combinators but opting for dimap can produce much more efficient code.

```haskell
class Category p where
  id :: p a a -- Maps a value to itself with no effects
  (.) :: p b c → p a b → p a c -- composition

class Category c ⇒ Arrow p where
  arr :: (a → b) → p a b -- arrows do at least what functions can do.
  (>>>) :: p a b → p b c → p a c -- also composition
  first :: p a b → p (a,c) (b,c)
  second :: p a b → p (c,a) (c,b)
  (***) :: p b c → p b' c' → p (b,b') (c,c')
  (&&&) :: p b c → p b c' → p b (c,c')

class Arrow p ⇒ ArrowChoice p where
  left :: p a b → p (Either a c) (Either b c)
  right :: p a b → p (Either c a) (Either c b)
  (+++) :: p a b → p b' c' → p (Either a b') (Either b c')
  (|||) :: p a c → p b c → p (Either a b) c

class Profunctor p where
  dimap :: (a' → a) → (b → b') → p a b → p a' b'
```

Listing 5.16: Haskell typeclasses related to the notion of `Arrow`.

All implementations of the interfaces in listing 5.16 are expected to conform to some laws which formalize the intuitions described about the meaning of arrows. We describe the laws required for each typeclass to be valid in listings 5.17

```
-- | Laws that need to hold for any valid category implementation.
-- Right identity
f . id = f
-- Left identity
id . f = f
-- Associativity
f . (g . h) = (f . g) . h


-- | Laws that need to hold for any valid arrow implementation.
arr id = id
arr (f >>> g) = arr f >>> arr g
first (arr f) = arr (first f)
first (f >>> g) = first f >>> first g
first f >>> arr fst = arr fst >>> f
first f >>> arr (id *** g) = arr (id *** g) >>> first f
first (first f) >>> arr assoc = arr assoc >>> first f

where
```

```haskell

assoc ((a,b),c) = (a,(b,c))

-- | Laws that need to hold for any valid arrow choice implementation.
left (arr f) = arr (left f)
left (f >>> g) = left f >>> left g
f >>> arr Left = arr Left >>> left f
left f >>> arr (id +++ g) = arr (id +++ g) >>> left f
left (left f) >>> arr assocsum = arr assocsum >>> left f

where

assocsum (Left (Left x)) = Left x
assocsum (Left (Right y)) = Right (Left y)
assocsum (Right z) = Right (Right z)

-- | Laws that need to hold for any valid profunctor implementation.
dimap id id ≡ id
dimap (f . g) (h . i) ≡ dimap g h . dimap f i
```

Listing 5.17: Laws for the typcalsses related to arrows.

Examples

To make the concepts of arrows and profunctors more tangible consider the following examples:

The Kleisli arrow The `Kleisli` arrow [20] as we mentioned earlier, is equivalent to the arrow `Monad m ⇒ a → m b`. In haskell it is defined as:

```haskell
newtype Kleisli m a b = Kleisli (a → m b)
```

So the arrow `Kleisli (State s) a b` is an arrow that when mapping values from `a` to `b` also operates on a state value of type `s`.

The state arrow We already saw the `State` monad in the previous section. If we expand the `Kleisli (State s) a b` arrow:

```
Kleisli (State s) a b
  ~= a -> State s b
  ~= a -> (s -> (b,s))
  ~= (a,s) -> (b,s)
```

The final way of putting it skips a few intermediate steps, we define a `StateArrow` then as

```
newtype StateArrow s a b = StateArrow ((s,a) -> (s,b))
```

Which is equivalent to the Kleisli arrow we created before: an arrow that when mapping values from `a` to `b` also operates on a state value of type `s`.

The `StateArrow` is not unique in its equivalence to a Kleisli arrow. This is a property that a couple of useful arrows have and that we will be exploiting in the following sections. Below is the interface we provide to describe the Kleislifieable arrow (named `ArrowFunctor` below) and a few examples

```
-- | Some arrows correspond to Kleisli arrows. We should be requiring
-- a monad functor but
class Functor (ArrFunctor c) => ArrowFunctor c where
  type ArrFunctor c :: * -> *

  toKleisli :: c a b -> a -> ArrFunctor c b
  fromKleisli :: (a -> ArrFunctor c b) -> c a b

-- | A kleisli arrow is trivially a kleisli arrow
instance Functor m => ArrowFunctor (Kleisli m) where
  type ArrFunctor (Kleisli m) = m
  toKleisli (Kleisli c) = c
  fromKleisli = Kleisli



-- | A state arrow is a state arrow with a bit of rearrangement.
```

```haskell
instance ArrowFunctor c => ArrowFunctor (StateArrow s c) where
  type ArrFunctor (StateArrow s c) =
    (StateT s (ArrFunctor c))
  toKleisli (StateArrow c) a = StateT $ \s -> swap <$> toKleisli c (s,a)
  fromKleisli f = StateArrow $ fromKleisli $ \(s,a) -> swap
    <$> runStateT (f a) s

-- | A writer arrow is also a kleisli arrow on top of a writer monad
-- with a bit of rearrangement.
instance ArrowFunctor c => ArrowFunctor (WriterArrow w c) where
  type ArrFunctor (WriterArrow w c) =
    WriterT w (ArrFunctor c)
  toKleisli (WriterArrow c) = WriterT . fmap swap . toKleisli c
  fromKleisli c = WriterArrow $ fromKleisli $ fmap (fmap swap . runWriterT) c
```

5.2.5 Mealy arrows: processes that remember

The concepts that we will dub a *mealy arrow* or *mealy process*, is kin to many different ideas: transducer, automaton and a coroutine to name a few. A mealy arrow (listing 5.18) is a function that along with the result gives a new version of itself. Like an *automaton* after every iteration it moves to a new state ready to continue the process. Like a *transducer* every element of a stream consumed changes a hidden state to be taken into account when consuming the next one. Like a *coroutine* it can be conceptualized as a process that can yield computation to be resumed by the caller.

```haskell
newtype MealyArrow a b =
  MealyArrow (a -> (MealyArrow a b,b))
```

Listing 5.18: Haskll definition of a mealy arrow.

The mealy arrow is at the heart of Antisthenis as they represent the basic building block of computation. We call the arrow returned by a Mealy arrow along with the value an *iteration*. Assuming that a mealy arrow represents an incremental computation, every iteration is produced in such a way as to exploit information about the structure of the problem based the previous computation.

For example a mealy arrow calculating a multiplication of several sub-arrows may want to try the ones that evaluated to zero in the previous iteration in order to exploit possible domain knowledge that nodes that are zero are likely to remain zero in future iterations.

Mealy arrow transformer

Like monads [8] , arrows can be composed together whey they are represented as *arrow transformers* For brevity we will only talk about mealy arrow transformers but the concept is generalizable to many different kinds of arrows [59].

The motivation behind "transformizing" the mealy arrow is that arrow evolution as in the plain mealy arrow we presented must be pure (due to it being facilitated by →). As we will see in detail when describing the internals of Antisthenis, as subrocesses evolve they need to interact with external data structures, may throw irrecoverable errors, need to run monadic computations from external libraries etc. None of that can be done based on a pure haskell function. For that reason we change the mealy arrow definition slightly (listing 5.19)

```haskell
newtype MealyArrow c a b =
  MealyArrow (c a (MealyArrow a b,b))
```

Listing 5.19: A MealyArrow can take on the properties of other arrows by swapping out the function → type for a parametric one.

Here the mealy arrow is parameterized by an arbitrary arrow `c` which may be a Kleisli or whatever arrow which will define what side effects are supported during the evolution of the process.

Building mealy arrows

While arrows are powerful and general, they can be slightly awkward to work with even with the haskell `proc` syntactic sugar. Transferring state between mealy iterations via their closures even more so. Capitalizing on the parallels between an arrow and a coroutine, and taking advantage of the rich haskell ecosystem around monads we define a monad `MB a b m` to help us build mealy arrows more easily. The interface to that monad is presented in listing 5.20.

```haskell
yieldMB :: Monad m ⇒ b → MB a b m a
mkMealy :: (ArrowFunctor c,Monad (ArrFunctor c))
```

```
          ⇒ (a → MB a b (ArrFunctor c) Void)
          → MealyArrow c a b
```

Listing 5.20: .The `MB a b m` monad can be used as a convenience to implement mealy arrows using a conroutine-like interfaces.

The important part is that we want to do something before each iteration, something after, and we want to feed some state to the next iteration like presented in listing 5.21, where it is how how we can transfer variables between iterations via the closure of the `do`-block.

```haskell
foo :: ArrowFunctor c ⇒ MealyArrow c a b
foo = wrapMealy arr $ \a it → do
  doSomeStuff
  (it',r) ← lift $ toKleisli (runMealyArrow it) a
  a' ← yieldMB r
  return it'
```

Listing 5.21: An example the usage of the MB functor to generate mealy arrows.

This involves `toKleisly` in reference to the "kleislifiable" arrows we saw in the previous section and lifting to `MB a b m` arrow which is precisely defined in 5.22. The basis here is the functor `MealyF a b` which structurally looks like a non-recursive version of a mealy arrow we saw earlier. That is if `x` is replaced with `MealyF a b \_` we get a mealy arrow with an extra value. Unfortunately there is no direct way to make a law abiding monad out of the `MealyF` type but there is a trivial functor for it, which allows us to use the free monad [23] to get the monad we want. Note how the Mealy arrow is *not* kleislifiable itself. The `MB a b m` monad can't produce any mealy arrow. We can produce only arrows that are from `a` to `b` from `MB a b m Void`.

```haskell
-- | Constructing monadic xmealy arrows
newtype MealyF a b x = MealyF (a → x,b)
type MB a b = Free (MealyF a b)
```

Listing 5.22: Definition of the MB monad transformer.

5.2.6 Zippers

Now that we have a fairly coherent view of computations we need one more piece of background knowledge to start putting together Antisthenis: the zipper data structure. A zipper [10] is an auxiliary data structure that is similar to (but not exactly the same as) a cursor.

For our purposes a zipper is a datastructure that represents an alternative configuration of a container such that it focuses on a specific location in that container, allowing an interface for reading and modifying the element at that location. A zipper should also be able to shift focus to adjacent elements.

Zippers can be very complex data structures with interesting categorical properties arising from the fact that they can implement a comonad interface [18] but for our purposes we only need a rudamentary understanding of it.

As an example consider the structure `ListZipper`:

```haskell
data ListZipper =
  ListZipper
  { lzLeft :: [a]
   ,lzCur :: a
   ,lzRight :: [a]
  }
```

A list zipper breaks the list in three parts: a left part `lzLeft`, a focused element `lzCur` and a right part `lzRight`.

```haskell
-- Make a zipper from a non-empty list
mkZipper :: [a] → Maybe (ListZipper a)
-- get a list from the zipper
getList :: ListZipper a → [a]
-- Focus on an element on the left if there is one
moveLeft :: ListZipper a → Maybe (ListZipper a)
-- Focus on an element on the right if there is one
moveRight :: ListZipper a → Maybe (ListZipper a)
-- Modify the current element
modCur :: (a → a) → ListZipper a
-- Read the element in focus.
getCur :: ListZipper a → a
```

These are fairly self-explanatory, a zipper can shift focus from the element in question to the neighboring elements.

5.3 Antisthenis core

In this section we discuss the core components of antisthenis. The parts that comprise antisthenis are roughly split in two categories.

- The core components that are related to building the antisthenis processes, the cells of computation, and the interfaces that need to be implemented to specialize them to specific operations.
- and the external that are generally the components tha relate cells to each other and the systems that specialize proecesses to implement specific operators.

While in this section we focus on the core components we will frequently alude to external components.

5.3.1 Processes

As we described in section 5.2.5 on mealy arrows define the basis of an antisthenis process. Indeed our definition of an antisthenis incremental cell is a composition of `MealyArrow`, `WriterArrow` and `Kleisli` presented in 5.23

```haskell
type ArrProc w m =
  MealyArrow (WriterArrow (ZCoEpoch w) (Kleisli m))
  (Conf w) (BndR w)
```

Listing 5.23: The type of an antisthenis process is a mealy arrow paired with a kleisli arrow.

The `w` parameter is simply a token related to the operation that the cell is evaluating that we use helps us parameterize the types involved in a process by leveraging the type family feature of haskell's type system [71]. We call it a `ZipperParams` token beacuse, as we will be seing thoughout the chapter, it mainly parameterize the evolution of the internal state of the antisthenis process. We will be looking in detail into the methods that `ZipperParams` tag needs to implement in the following sections.

We will describe every type involved in the `ArrProc` type separately, but to do that we first provide a general intuition of what these types mean. The basis is a mealy arrow that emits monoidal value via `WriterArrow`, along with the values, we dub `ZCoEpoch`. As we will see later in more depth (section 5.3.3 on Antisthenis caps sand bounds), it is a datastructure that summarizes the provenance of the value returned. It is used to determine when the value is no longer valid due, for example, to changed parameters.

The `MealyArrow` and `WriterArrow` combined transform an arbitrary Kleisli arrow which is used to embed the computation of antisthenis processes into parent monadic computations. In the FluiDB case that would be the monad computation assocuiated with planning. This way we give the antisthenis processes the means to lookup values, throw errors, or generally have effects of any computation we require.

The output side of the `ArrProc` is a `BndR w` type, the returned value, which is parametric to the operation but has some common structure (listing 5.24). This indicates that every Antisthenis process has three options for the value they can return all of which are parameterized on a per-operation basis (ie `ZErr`, `ZBnd` and `ZRes` are type families):

- An error indicating that the value is not computable.
- A final and precise result.
- A partial result, a convex bound indicating that calling the iteration of the arrow will get us closer to ax‘ final result. We will have a chance to see the bounds in detail in the section 5.3.3 oncaps and bounds.

```haskell
data BndR w
  = BndErr (ZErr w)
  | BndRes (ZRes w)
  | BndBnd (ZBnd w)
```

Listing 5.24: The definition of the return value of an Antisthenis process. It may be a final result, an error indicating that a final result is non-computable, or a bound for the final value.

Indeed, the iteration of the process has a very different meaning depending on whether the previous returned value is a bound or an error/result. In the former case the iteration continues a previous computation. In the latter case there are two options. Eitger the previously returned value is still valid and is therefore returned, or if it is not and the computation is reset. To reflect this fundamental difference between a paused and a finished computation we will call the former simply an *iteration* and the latter *coiteration*.

Finally, we turn our attention to `Conf w` which represents the configuration for running the next step of the computation. Much like `BndR` the configuration is parameterized w.r.t. `w` but has a basic structure presented in listing 5.25

```
data Conf w =
  Conf { confCap :: Cap (ZCap w)
        ,confEpoch :: ZEpoch w
       }
```

Listing 5.25: The type definition of a conficuration. It is a tuple containing information that can be used to derive whether a value is valid.

The configuration for running a computation includes an epoch (introduced in section 5.26) and a cap (introduced in 5.3.3). It is passed as input to each computation and propagated to the children computation. If the process is iterating (as opposed to co-iterating, i.e. it has not returned a concrete result or error) the cap is compared against the previously yielded bound, if the cap is less than the bound returned then the process must keep processing until it found a bound that exceeds that cap. This way we can control how much work a process is allowed to do before switching to one that might help Antisthenis to cut the overall computation short. This way Antisthenis avoids falling into computational rabbit holes, where it evaluates a long computation when switching to a shorter one would help arrive sooner to a final result.

The epoch represents state external to Antisthenis like parameter versions. When the epoch indicates that a parameter on which the process depends has been updated, the process needs to be reset and start computation form scratch (see section 5.3.6 on process resetting).

5.3.2 Epochs and coepochs

As mentioned in the previous section the interplay between epochs and coepochs determine the validity of the value of each Antisthenis process. When the retuned value is no longer valid the process needs to be reset. In particular, the epoch mainly indicates the "version" of the computation parameters. This might represent the full external state, but the purpose of the epoch is to be used as an increasing value that can be used by the processes to infer whether their progress is based on out-of-date assumptions. For example the epoch may be a map of a natural number per computation parameter indicating the number of times that parameter has been updated. Or it might simply be the actual value of each parameter if it is cheap the compare against. Each process then can compare the version or value of each parameter of the epoch to the ones it used to build its internal state. When a discrepancy is found the process knows its state is out of date and must reset.

It is clear that not all parameters are relevant to all processes. Most processes only depend on a subset of parameters. In Antisthenis' terms, most processesonly ever depend on parts of the epoch. This reference to a subset of any epoch is reified by the coepoch. Coepochs are the monoidal type parameter of the writer arrow transformer of `ArrProc`. In practice this theman that due to the writer semantics the coepochs of evaluated subproecesses are concatenated to produce the coepoch of the parent process. A special case for this are the commutative operations that involve absorbing elements ($\wedge$, $\vee$, and $\times$), where only the coepoch of the subprocess that yields an absoring element is returned. In short, the coepoch of a subprocess is included in all parent processes that are not constant with respect to the value of that subprocess.

In principle the function that the `ZipperParams` tag needs to implement with respect to the epoch/coepoch interplay is a function that has a type equivalent to.

```haskell
combEpochCoepoch0 :: ZEpoch w → ZCoEpoch w → Bool
```

Listing 5.26: The type of a naive function checking the validity of a value based on epoch and coepoch.

That will indicate whether the value needs to be pushed down. In practice we can do slightly better at that by enforcing the following law on the `combEpochCoepoch0` function:

$$\begin{aligned}&\text{combEpochCoepoch}_0(e, c_0 \diamond c_1) \Rightarrow \\ &\text{combEpochCoepoch}_0(e, c_0) \wedge \text{combEpochCoepoch}_0(e, c_1)\end{aligned}$$

Where $e \in ZEpochw$, $c_0, c_1 \in ZCoEpochw$ and $\diamond$ is the monoidal merging operation on coepochs.

We therefore have the option of filtering the epoch into a smaller subset of itself that is only relevant to the subprocesses that contributed to the generation of the coepoch being checked. Thereby the combination function can optionally return a new epoch rendering the type of the function to be something similar to listing 5.27. It is worth noting that if by this process we determine that a reset is required, a new coepoch is required and therefore there is no way to constrain the coepoch for the reset. In the example above if `coepoch0 <> coepoch1` do not match the epoch, it is possible that re-evaluation of P_1 will not yield `BndRes 0` and thereby we would need to evaluate P_2 including its coepoch to the aggregate.

```haskell
combEpochCoepoch :: ZEpoch w → ZCoEpoch w → Maybe (ZEpoch w)
```

Listing 5.27: The final function for epoch and coepoch combination.

5.3.3 Antisthenis caps and bounds

In the section regarding epochs and copeochs we described how Antisthenis deals with the validity of values returned by a process. In this section we will focus on how the children processes avoid falling into computational rabbit holes using the interplay of *caps* and *bounds*. In the example discussed in the introduction (5.1), we saw that there are case where, as soon as a subprocess can prove that its value is going to exceed a certain threshold (*cap*), any further work on its part is likely futile. It then returns a *bound* which is propagated up the chain of parent processes and handled at the point where the cap was imposed. The reader is encouraged to maintain a mental model where the cap is an threshold and the bound is a lower bound.

We expect that the bound of a function satisfies the *mototonicity criterion*:

$$C \dot{<} B[f(a_0, ...)] \Rightarrow C \dot{<} f_B(B[a_0], ...)$$

Where C is the cap, $B[E]$ is the bound of a process expressed by the expression E by only taking into account the top level terms. f is the function if the operator and f_B is a function that combines the bounds of the arguments to come up with a bound for f. $C \dot{<} b$ means that the bound b exceeds the cap C.

In simpler terms this means that if a bound exceeds the cap, evaluating the arguments and coming up with a tighter bound will still exceed the cap.

There may be more requirements of the properties of a cap relate to the particular operator. For example the minimum operator requires that bounds be translatable to caps and that if a bound b is translated to a cap c and $b' \dot{<} c$ then $b' < b$. This requirement is only related to the way the particular operator is implemented and is not a requirement for any other operator.

The `ZipperParams` tag needs to provide a way for comparing caps to bounds. Unlike the case of coepoch and epoch combination function this one is much more straightforward:

```
exceedsCap :: ZCap w → ZBnd w → Bool
```

Simply from a cap and a bound check if the bound exceeds the cap.

Now that we have seen how the cap is handled on the input side we should discuss how the parent process passes a cap to its child processes. The cap is created in an operator-specific way (ie via the overloading of a function based on the tag `w`). In particular an operator defines a function that ”localizes” the entire configuration.

```
data MayReset a = DontReset a | ShouldReset
zLocalizeConf :: ZCoEpoch w → Conf w → Zipper w p → MayReset (Conf w)
```

The `zLocalizeConf` function accepts a coepoch, the configuration that the parent process receives and returns the configuration passed. The `zLocalizeConf` function alsoc decides whether the process needs to be reset by calling the `combEpochCoepoch` function that we discussed in subsection 5.26 on epochs and coepochs.

To make all this more tangible, a process that adds epoch implements the `zLocalizeConf` roughly like shown in listing 5.28.

```
zLocalizeConf coepoch conf z =
  combEpochCoepoch coepoch (confEpoch conf)
  $ conf { confCap = newCap
         }
  where
    -- zRes is the partial result so far
    newCap = case zRes z of
      SumPartErr _ → error
        $ "The partial sum is an error this thunk "
        ++ "should never be reacahble beacuse "
        ++ "no subprocesses should be called."
      SumPartInit → confCap conf -- No subprocesses have been evaluated.
      SumPart partRes → case confCap conf of
        -- partRes is the total sum so far. We offset the global cap
        -- by the sum so far so that the local cap ensures that the
        -- cursor process, when ran, does not cause the overall
        -- bound to exceed the global cap. This may generate weird
        -- caps like negative values. That is ok as it should be
        -- handled by the evolutionControl function. Note again that
        -- zRes does not include the value under cursor.
        CapVal cap -> CapVal $ subCap cap partRes
        ForceResult -> ForceResult
```

Listing 5.28: A sample implementatation of the function that transforms the configuration received by a parent process into one suitable for the child process. Checks if the parent process needs to be reset and uses the partial result to constrain the cap.

Since the process knows it will be evaluating the cursor, it subtracts the partial sum so far from the cap to come up with the cap to pass to the subprocess.

5.3.4 Antisthenis zipper

We saw in the introduction on Antisthenis (5.1) what a zipper is in general. Here we will specialize the notion of a section for the specific case of the internal state of the computation. A single process is defined by an operator, the subprocesses and a partial result for the computation. The process evolves by evaluating a subprocess at a time. We define a zipper structure that focuses on a the particular process to be evaluated and arranges the reset of the subprocesses in such a way that moving around in the datastructure we focus on the next subrpocess to be evaluated. The internal structure of the zipper helps antisthenis decide on the next subprocess to be evaluated.

With this in mind we define the zipper data structure keeps track of the state of the node (see figure 5.2).

- A set of `initial` processes that have not been evaluated yet or that have yielded deprecate values (see section 5.26 on epochs and coepochs).

- A set of `iteration` processes, processes whose latest evaluation has yielded a result bound is represented by a data strucutre that associates the iteration processes with `initial` processes the bounds returned. Since these processes that have yielded a e and need to be rerun with a different cap (see section 5.3.3). It should be stressed that this still acts like a heap of subprocesses where the internal properties of the data structure decide the top element that is to be popped. Since we have some information about the final result of these processes, namely the bound, the parent process can be smart about the order in which they are evaluated. As the particular strategy is dependent on the operation implemented by the parent process, the particular data structure used is also dependent on said operation. For example a process implementing a sum between the subprocesses stores them in a list as there is no beneficial order in which to evaluate the subprocesses while minimum openration benefits from evaluating the processes that have lower bounds first (for more details on the minimum oparation see section **??**).

- Coiteration processes are processes that have been evaluated and yielded a concrete e (error or result), ie a value that is predicated on the epoch (see section 5.26) but can not be refined by re-evaluating the process with a different cap . Upon reset the coiteration stack is ordered in an operator specific way and moved to the init stack.

- A cursor that is the next machine to be triggered along with a way to reset it and the previous value

- A partial value to which values can be inserted or removed. The partial value does *not* include the value stored in the cursor.

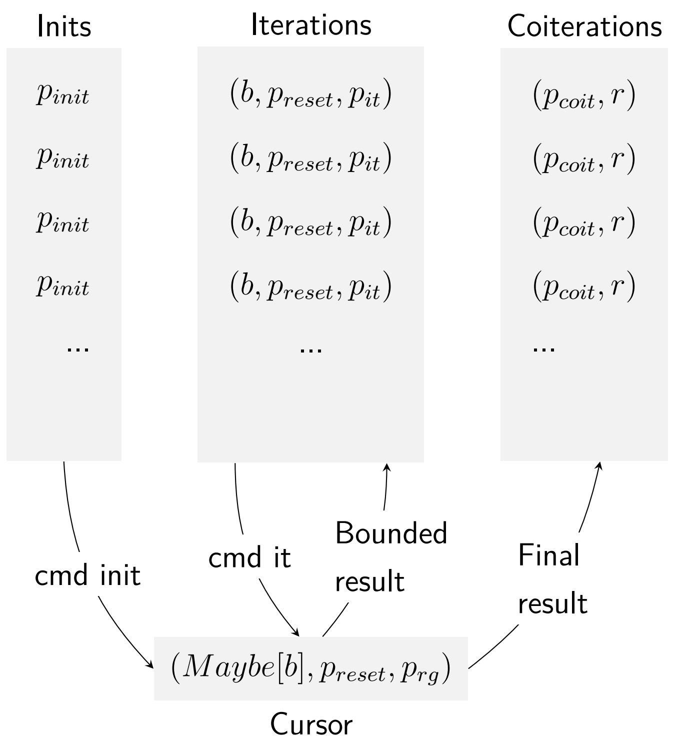


Figure 5.2: The zipper splits the sub-processes that cooperate to compute values for the owning process in four categories a) initials that have not yielded a currently valid value b) iterations that have not yielded a valid partial value c) coiterations that have yielded a full valid value d) the cursor that is the next sub-process to be evaluated. Depending on the evaluation strategy of the operator once the cursor process evaluates it is replaced with an init process or an iteration process, and depending on the value the cursor process yields it is pushed to the iterations or the coiterations of the zipper.

During computation, the subprocesses owned by the parent process are evaluated, and their iteration are then moved from the `initials` set to the `coiterations` set, with an intermediate stop in the `iterations` set according to the values and strategies. These are further elaborated in the section on the section antisthenis operations. The structure of `iterator set` is highly operator dependent and primarily geared towards efficiently evaluating the parent process, but fundamentally it does not compromise on the correctness of the operator. For example the if a process is combining elements of an abelian group (commutative, associative, invertible) and the bound has the same type as the result – as it is the case with addition – the structure of the zipper is fairly straightforward:

- the partial result maintains the aggregation of all values and bounds encountered
- When evaluating the cursor we add the result to the group and remove the bound of the next cursor if we are popping from the iteration set.
- The the partial result is precisely the bound of the final value.

At the other end of the spectrum, a process evaluating a magma (a set with a closed binary operation), where the binary operator has no properties besides closure, must never push anything to the iterations set. The set is equivalent to unit.

As promised we will expand on these in section 5.4 on antisthenis operators but to make the concept tangible here are some examples of what the iterator set is defined as for different operations (all numerical opertions are assumed to operate on non-negative numbers, in particular query costs):

- Addition is the simplest case: the iteration set is an always-empty container since we always evaluate the cursor until it is sent to the coiteration list. In the context of addition there is no heuristic to predict which subprocess is more beneficial to evaluate.
- For multiplication over positives we want to evaluate the ones that have a zero bound first in the hopes that they will turn out to evaluate to absorbing elements. Therefore, the iteration set is represented as two lists: one where all the zero-bounded iterations are stored and one for all the rest.
- In the case of minimum the iteration set is a heap. We always want to evaluate the one with the minimum bound. When the minimum bound exceeds the minimum concrete result encountered, the min result is the final result.

For completeness we provide the definition of the `Zipper` in listing 5.29

```haskell
data ZipState w a =
 ZipState
 { bgsInits :: [InitProc a]
  ,bgsIts :: ZItAssoc w (InitProc a,ItProc a)
  ,bgsCoits :: [(Either (ZErr w) (ZRes w),CoitProc a)]
 }

```

```
data Zipper' w cursf (p :: *) partialRes =
  Zipper
  { zBgState :: ZipState w p
   ,zCursor  :: cursf (Maybe (ZBnd w),InitProc p,p)
   ,zRes     :: partialRes -- The partial result without the cursor.
  }
type Zipper w p = Zipper' w Identity p (ZPartialRes w)
```

Listing 5.29: The definition of the zipper.

Finally, the operator defines the partial result type of the zipper `ZPartialRes` and functions for putting and replacing values in it. When the subprocess on the cursor is evaluated the result needs to be either "appended" into the partial result if the new cursor is drawn from the initials set, or to replace the corresponding bound value if it is replaced with a value from the iterations set.

5.3.5 The `Cmds` functor

As indicated in the section on zippers the process evolution is guided by moving the zipper focus to different subprocesses and then evaluating them.

A process is internally represented as a mealy arrow (introduced in section 5.2.5) describing the evolution of a zipper (introduced in section 5.3.4). An operator is allowed to follow a strategy defined specifically its `ZipperParams` tag, but the function for evaluating said strategy applies to the `ZProc` object. There are two important aspects to be noted about `ZProc` (listing 5.30).

- The return value is a zipper full of processes `Zipper w (ArrProc w m)`
- The Kleisli arrow is a free functor of commands `FreeT (Cmds w) m`.

```
type ZProc e m =
  MealyArrow
    (WriterArrow (ZCoEpoch w) (Kleisli (FreeT (Cmds w) m)))
    (LConf w)
    (Zipper w (ArrProc w m))
```

Listing 5.30: An internal representation of the process evolving the internal representation of a process: the zipper.

`Cmds` is a union type of different directions toward which the zipper can be evolved. Its definition is provided in listing 5.31. In this general an operator agnostic form the evolution can take one of a few state transitions at a time:

- It can always be reset
- It can pop a process from the initial to the cursor to be executed
- It can pop a process from the iterator set if it is not empty. Which one is to be popped is decided by the parameters
- The may have reached a final value where it can't be evolved anymore.

```haskell
data Cmds' r f a =
  Cmds { cmdReset :: ResetCmd a
        ,cmdItCoit :: ItInit r f a
       }

data ItInit r f a
  = CmdItInit (ItProcF f a) (InitProc a)
  | CmdIt (ItProcF f a)
  | CmdInit (InitProc a)
  | CmdFinished r -- when a process finishes it should stick to a
                  -- value until the epoch/coepoch pair requests a
                  -- reset.
```

Listing 5.31: Definition of the commands functor that provides different branches of evolitution for zipper.

So the `Cmds` facilitates a tree of actions in conjunction with a free monad . We have talked about the free monad briefly, to recap a free monad of functor `f` is actually many `f` nested like `f (f (f ... f (f a)))`. This means that `FreeT Cmds m` is an arbitrary depth tree of `Cmds`. This tree is navigated in an operator specific way guided by the generic function `evolutionStrategy` the type of which is presented in listing 5.32.

```haskell
evolutionStrategy
  :: forall x .
  FreeT (Cmds w) m x
  → m (Maybe (RstCmd w m x),Either (ZCoEpoch w,BndR w) x)
```

Listing 5.32: A function that each atnistenis opator needs to implement that decides the traversal of the tree created by the different possible evolutions of zipper. The implementation may also deem that a good reset point has been discovered.

This function navigates the tree in the way that is most beneficial to the particular operator and comes up with two values:

- A reset command to be used if reset is required in the future (this will be discussed in the section about 5.3.6 resetting)
- When encountering a `CmdFinished` value return the result of the finished operation, otherwise traverse the tree to the preferable leaf and return the leaf.

An abridged example is presented in figure 5.3.

5.3.6 Resetting processes

In the subsection discussing the command functor we saw that it may include a reset process. Why create a reset process when reusing the initial process would reset the computation just as well? In any case a reset moves all the init parts of the elements of iteration set as well as the coiterations into the init set of the zipper. But in what order? From previous runs it is often apparent that some execution strategies are more likely to lead to a quick result than others. For example, under the assumption that processes change their values less often than they maintain them, while computing a logical AND operation ($\wedge$) we likely want to start the computation from processes that previously yielded `False`.

To accommodate this, the `Cmds` functor (introduced in section 5.3.5) can yield a reset point. While running a process we maintain in the lexical scope of the mealy arrows computation the latest reset process to switch to when the epoch/coepoch comparison requires as much. When generating a `Cmds` functor, the operator has the option to implement the `bgsReset` that resets a running zipper that is at an arbitrary state. The signature and default implementation are below:

```
bgsReset :: ZipState w a → ZipState w a
bgsReset bgs =
  mkGgState
  $ bgsInits bgs
  ++ (snd <$> toList (bgsIts bgs))
  ++ (snd <$> bgsCoits bgs)
```

5.4 General operators

While describing the core concepts of Antisthenis we often dealt with functionality that is operator specific. Indeed, a large part of Antisthenis depends on operator specific logic. We define operators at three levels, each refining the previous. The lower levels are parameterized by the higher ones.

The lowest level is reified as an uninhabited type, or tag, witnessing the implementation of `ZipperParams`. This implementation would be build a workable Antisthenis type. The operator specific types are presented in listing 5.33. We have implemented a few types in antisthenis to cover the needs of FluiDB (see listing 5.34).

```
-- Result types
class BndRParams w where
  -- The result type when the value is non-computable
  type ZErr w :: *
  -- The partial result.
  type ZBnd w :: *
  -- The full result
  type ZRes w :: *

-- Zipper internal types
class BndRParams w ⇒ ZipperParams w where
  -- The type of the process cap
  type ZCap w :: *
  -- The type of the process epoch
  type ZEpoch w :: *
  -- The type of the coepoch
  type ZCoEpoch w :: *
  -- The type of the iteration set
  type ZItAssoc w :: * → *
  -- The type of the partial result
  type ZPartialRes w :: *
```

Listing 5.33: Operator specific types that need to be impolemented by every operator.

```
-- The witness type for the sum operator
data SumTag p
-- The witness type for the minimum operator
data MinTag p
-- The witness type for multiplication
data MulTag p
-- The witness type for logical And/Or
data BoolTag op
```

Listing 5.34: Tags are phantom types that define the antisthenis operations. The tags themselves may be parameterized using type parameter `p`. For the needs of FluiDB we define operations for addition, subtraction, multiplication and boolean operations.

The parametric part of the tags, especially for `SumTag` and `MinTag`, is due to the fact that we want to be able to further parameterize them for concrete cost calculation (using sum and min operators normally) and for stochastic cost calculation (see section on 5.5.4 historical cost). These two both utilize minimums and sums, but they use different types for return values as well as caps/bounds and epochs/coepochs.

Since all types and implementations can be derived from the `ZipperParams` type `w`, the Antisthenis process creation function can be as simple as in listing 5.35, where `mkProc` combines a list of subprocesses into a parent process that evaluates the operator described by the tag `w` without requiring any further information.

```
mkProc
  :: (Monad m,ZipperParams w,Eq (ZCoEpoch w)) ⇒ [ArrProc w m] → ArrProc w m
```

Listing 5.35: The type `w` fully defines the operator so combining subprocesses into a process is unumbiguous.

Compatibility between heterogeneous processes

The astute reader has likely noticed that `w` is the parameter of both the subprocesses and the process. The reason is that a process expects to see a specific type of coepoch and bound returned by its subprocesses, while the subprocess must be able to deal with the epoch and cap that the parent process is defined for. To do that we avoid defining an $N \times N$ correspondence between all the epochs, but rather we define ad hoc a record that facilitates the conversion, the type of which is presented in listing 5.36. This way we can compartmentalize the translation among operators from the internal logic of the operators.

```haskell
data Conv w w' =
  Conv
  { convEpoch :: ZEpoch w' → ZEpoch w
   ,convCoEpoch :: ZCoEpoch w → ZCoEpoch w'
   ,convCap :: ZCap w' → ZCap w
   ,convRes :: ZRes w → ZRes w'
   ,convBnd :: ZBnd w → ZBnd w'
   ,convErr :: ZErr w → ZErr w'
  }
convArrProc :: Monad m ⇒ Conv w w' → ArrProc w m → ArrProc w' m
```

Listing 5.36: An object of type `Conv w w'` can act as an interface between parent processes of type `w` to subprocesses of type `w'`.

5.4.1 Basic cost operators

General cost refers to minimum and sum operators with some special semantics:

- Non-computable values are larger than any value from the perspective of minimum.
- All values are positive (therefore all values are implicitly bounded by $[0 - \epsilon)$

The minimum operation computes the minimum of the argument operations.

- Iteration data structure of the minimum is a heap-like structure where we can peek at the top element and look into the secondary one and it also contains the value derived from the s encountered.
- Zipper cursor always contains the most promising.

Sum

We do not implement a general sum operator but rather a sum operator that assumes the properties of query costs as the values it deals with. The sum is probably the simplest operator implemented. The adding zipper evaluates each initial subprocess under a cap $c_{subproc}$ that is

$$c_{subproc} := c_{parent} - r_{partial}$$

where $r_{partial}$ is the partial result omitting the cursor, ie the current subprocess and c_{parent} is the cap that the parent process is operating under. If the parent cap is **ForceResult** all arguments are also evaluated under **ForceResult**. Of course if $c_{parent} > r_{partial} + b_{cursor}$ then $r_{partial} + b_{curopr}$ is returned as the result bound.

Multiplication

Multiplication is not used by FluiDB but is implemented in Antisthenis nonetheless as a reference because it is fairly simple and yet it implements exploits many of the features of Antisthenis. The multiplication goes through all the initial subprocesses capping them at zero until one of them returns a concrete result of zero in which case the final result is also zero. Because a bound returned by a process is strictly greater than the cap all the iterating results are guaranteed to not be absorbing elements. Then multiplication proceeds in the same way as addition does only multiplying instead of adding:

$$c_{subproc} := c_{parent} / r_{partial}$$

Minimum

Much like the sum operator, the minimum operator implementation also assumes that it is manipulating cost types, and it is used in conjunction with the sum operator to define our cost evaluations. The operation of the minimum Antisthenis operator depends heavily upon the iterator set which has the interface of a heap. The zipper evolution stategy is described by the pseudo-python code in listing 5.37

```python
def exceeds_cap(cap, zipper):
    return zipper.iter_procs.min().bound > conf.cap \
        and zipper.coiter_procs.min().result > conf.cap \


def minimum_zipper_evolution(conf,zipper):
    # Run each initial process with a zero bound
    while len(zipper.initial_procs) > 0:
        zipper.inter_procs.push(cursor)
        zipper.cursor = zipper.initial_procs.pop()
        res,nxt_proc = zipper.cursor.run(conf{cap = 0})
        # We cant hope to get a cost lower than zero
        if res == BndRes(0):
            # Until the epoch is updated
            return BndRes(0)

        # Push it to the correct iter set.
        if res.is_bound():
            zipper.iter_procs.push(res,nxt_proc)
        else:
            zipper.coiter_procs.push(res,nxt_proc)

    # Run the iterating processes
    while True:
        # If the result is better than the best iteration, it's the final result
        if zipper.coiter_procs.min().result < zipper.iter_procs.min().bound:
            return BndRes(zipper.coiter_procs.min().result)

        # If there are two iter processes
        if not exceeds_cap(conf.cap,zipper):
            # Run the best iterating process until it exceeds the
            # second best iterating process or the final result
            cap = min(zipper.iter_procs.second_min().bound,
                      zipper.coiter_procs.min().result)
            res,nxt =  zipper.iter_procs.min().run(conf{cap = cap})
            # Push it to the correct iter set.
            if res.is_bound():
                zipper.iter_procs.push(res,nxt_proc)
            else:
                zipper.coiter_procs.push(res,nxt_proc)
        else:
            conf = yield zipper.iter_procs.second_min().bound
```

Listing 5.37: Pseudocode for the algorithm of evaluating a process up to a threshold defined by the cap. For brevity and to avoid too much unnecessary detail we omit sanity checks and the reset handling code.

Boolean conjunctions and disjunctions

Antisthenis implements operators for boolean conjunction and disjunction in order to be able to answer the question of whether a node is materializable in the presence of a specific inventory of materialized nodes. Booleans are different to the other operations in how they handle caps. Boolean $\wedge$ and $\vee$ do not use the same type for caps and bounds as for results. Instead the bound is a two dimensional value that represents the minimum number of steps required to arrive to each possible value (`True` or `False`).

To efficiently evaluate boolean algebra expressions we make look for absorbing elements, which this time demonstrates how absorbing elements can be exploited to prune the expression tree, consider the problem of evaluating a Boolean expression $A \wedge B \wedge C$ where A, B and C are subexpressions. Obviously if any of the expressions A,B or C evaluates be `False` the entire expression would consequently evaluate to `False` as well as `False` is the absorbing element for the group of booleans over $\wedge$. Furthermore without taking into account the structure of each of the subexpressions A, B and C we can tell that the amount of work for evalutating it depends on the result. If the result of this expression is `False` then the best case scenario in terms of amount of work we would need to do is for A to be the expression `False`. In this case we would be able to evaluate the expression with just one dereference. If the result is `True` then the best case scenario is for A, B and C to all be `True` and we would be able to come to a result with three dereference steps.

Consider the expression

$$X := (A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2) \wedge (C_1 \vee C_2 \vee C_3)$$

Where A_i, B_i and C_i are expressions. How would we navigate this expression to minimize the number of operations? We could completely expand each term and hope that we are lucky, and we encounter absorbing elements early on. As explained in the introduction, especially for terms automatically generated this strategy is unlikely to be efficient.

Instead, at each expansion we count the minimum number of steps required for each value. In the above example the best case scenario if the value of X is `False` is $B_1 = False$ and $B_2 = False$, in which case the expression is evaluated in a minimum of 4 steps:

- Evaluate $B_1 = False$ in a single step
- Evaluate $B_2 = False$ in a single step
- Evaluate the subexpression $False \vee False$
- Evaluate $X = (...) \wedge False \wedge (...)$

For a value of `True` it is 7:

- Evaluate $A_i = True$ in a single step for any i
- Evaluate the subexpression $True \vee ...$
- Evaluate $B_i = True$ in a single step for any i
- Evaluate the subexpression $True \vee ...$
- Evaluate $C_i = True$ in a single step for any i
- Evaluate the subexpression $True \vee ...$
- Evaluate $X = True \wedge True \wedge True$

A boolean expression in Antisthenis, then, takes the bound value of the minimum cost for trues and falses. If this expression were evaluated with a cap of $\{True : 6, False : \infty\}$ it would return a bound of value $\{True : 7, False : 4\}$. In plain english this means "try to evaluate this if your other options are more expensive than 7 steps for `True` and 4 for `False`". It should be seen as a metric for the size of the tree.

To demonstrate the derivation of caps we present the following expression:

$$X = X_1 \wedge X_2 \wedge X_3$$

Any of the X_i terms has a minimum bound of $\{True : 1, False : 1\}$. Since the absorbin element for $\wedge$ is `False` expanding any of the X_i terms is bounded by $\{True : \infty, False : 1\}$ bacause so an expansion of

$$X_1 = A_1 \vee A_2$$

Would cause the subprocess to immediately yield $\{True : 2, False : 3\}$ passing control to the parent process. Thus, a symmetrically growing tree would be traversed in a breadth first fashion and any asymmetries would cause Antisthenis to try to greedily exploit the smaller parts of the tree.

Antisthenis has a different regime for comparing caps and between a cap and a bound. We consider that a bound exceeds a cap when either the steps to a true final value or the steps to a false final value exceed the corresponding cap. However when it comes to comparing bounds we consider the better bound to be the more optimistic one.

$$b_1 <_{bnd} b_2 := \min(T(b_1), F(b_1)) < \min(T(b_2), F(b_2)) c <_{cap} b := T(c) < b \wedge F(c) < F(b).$$

where for x being a bound or a cap, $T(x)$ is the minimum number of dereferences assuming the final value is `True` and $F(x)$ is the minimum number of dereference values assuming the final value is `False`.

5.5 Node machines

A referrable machine is a process with a name that can be referenced by more than one other processes. In FluiDB a machine name corresponds to a graph node since the antisthenis infrastructure is utilized to provide incremental computation of properties on nodes (materializability, cost, stochastic cost, etc are described in separate sections). In this section we look into the the layer of antisthenis that takes into account the dynamics of referrable machines.

5.5.1 Antisthenis machine tape

The machine tape is a dynamic directory that maps nodes to machines. The meaning of a node is dependent on the implementation, in the context of FluiDB nodes in this context correspond to the FluiDB internal graph nodes. The machine tape is part of the computation context and is updated by new and temporary values of machines (see section 5.5.3 on cyclic machines).

Not all processes involved in a computation are involved in the machine tape. Each machine in the tape is composed by a hierarchy of machines that are inaccessible from outside. In fact a machine is discriminated from a process solely by virtue of the fact that it can be referenced via the tape. A parallel might be drawn between machines as being like C++ lvalues in that they are referable by an address, and processes being like rvalues in that they are still values but they are completely nameless. Furthermore, as the astute reader will have noticed, it may not be clear in every context when a process stops being itself. Remember that a process is fundamentally a mealy arrow (introduced in section 5.2.5) which once evaluated returns a new mealy arrow, the next version of itself. Self-sameness is much more concrete in the case of machines, as machines are always referred by exactly one slot in the tape.

Since machines are referable by different other processes it means that, we need to be carful that more than one process do not own their own diverging copy of the same machine, essentially treating it as a process. To avoid this issue, processes referring to machines do so indirectly via a **machine wrapper process**. A machine wrapper is a process that does the exact same thing every time it is evaluated:

- it looks up a name in the tape
- it temporarily replaces it with a cyclical machine (see section 5.5.3)

The machine tape evolves externally to each individual process and each process evolves within the tape itself. We implement it inside a state monad in the underlying Kleisli arrow. Thereby a recursive reference is formed between the tape and the machine.

5.5.2 Process stack

Before we go further with the details of how exactly named machines are handled, it is important to make the discrimination between a process and a computation. In the context of Antisthenis a process, or in case it is named a machine, is the mealy machine that evolves during evaluation and survives between evaluations. On the other hand a computation is the part of the evaluation itself that refers to the particular machine. At the risk of getting too philosophical, a computation is the **relation** between the process and the value produced, regardless of whether that value is an actual result, a bound for a potential result or a witness to the non-computability of the result.

With this distinction in mind we introduce the notion of a process stack which is similar to the notion of a call stack in any programming language. The process stack contains all processes that have live computation. When a process is called, it is pushed to the process stack and when a result is returned it is popped from the process stack. The process stack is completely ephemeral, acyclic and should be completely empty between computations. In this sense it is part of the **computation context**, the sum of effects that affect each computation separately. The computation context may be implicit (like in the case of the process stack as we will see when discussing cyclical machines) or explicit in the process' underlying Kleisli arrow.

5.5.3 Cyclical machines

By virtue of machines being referable by arbitrary other machines, it is possible, and certain in the case of FluiDB, that the Antisthenis machines will form referential cycles. Such cycles in most computational frameworks, especially ones that deploy a non-lazy order of computation, will cause the computation to either fail or to not-terminate. Antisthenis is explicitly designed around allowing the computation to handle such self-referential cases.

We already discussed in a previous section that machines are stored in data structure called a tape that allows us to reference them, and that in the period from the time a value is requested of a machine and the time a value (including an error or a value bound) the machine is said to be in the computational stack (introducted in section 5.5.2). During its residence in the stack a machine is in a state where it is unable to produce a value. A cycle occurs when a machine is referenced while in the computational stack.

We make the problem concrete with an example based on the graph in figure 5.4a. A naive approach of dealing with recursive calls is demonstrated in figure 5.4c where naive-Antisthenis looks for cycles by throwing errors ($\perp$) when encountering a node that is already in the computational stack. Evaluating node A Antisthenis first node B to 10.

It then tries to evaluate node C which redirects back to A. C can therefore not yield a value and returns an error $\perp$. Then, without changing any of the parameters the user tries to evaluate D. Looking at the graph it is clear that D should in sequence evaluate to 15, as demonstrated by the evaluation process in figure **??** (the reader is reminded that based on the assumption that all values refer to query evaluation costs we have asserted that $min[a, \perp] \equiv a$).

The root of the problem in the naive case is that the process stack is an ephemeral property and yet it affects the return values that are non-ephemeral. The only solution for this is to lift information about the cycles to the non-ephemeral plane that survives the computation. Indeed, what is needed is for values that are deemed non-computable to *reset* when accessed with a different stack. In other words the value of $\perp$ for C is *predicated* on the process stack containing A, or A being otherwise non-computable. Coepochs fill the bill precisely. We extend the coepoch to include, along with the parameter versions used to derive a particular result, also a set of machines that need to be non-computable in order for the value to be valid.

To accomplish this when a machine is looked up, before being evaluated it is removed from the tape and in its place a **cyclic machine** is placed. This machine always returns the same $\perp$ result that indicates that the value is non-computable and sets to the coepoch that it is itself not computable. When the original machine comes up with a value it replaces the cyclic machine again and the coepoch is sensored. This way whenever any machine tries to evaluate another non-computable machine it finds a cyclic machine which handles the situation automatically. Upon evaluating the cyclic machine, the coepoch of the caller is infected with the predicate that the cyclic machine must be $\perp$. Pseudo-python algorithm in listing 5.38.

```
def cycle_machine(node):
    # Return a process that returns non-comp and a predicate nofifying
    # that it is itself not computable.
    class CycleProc:
        def run(conf):
            return (lookup_node(node),non_comp,Coepoch(predicates=[node]))

    return CycleProc()

def run_submachine(node,conf):
    # Replace the real process with a dummy cycle process.
    p = lookup_node(node)
    insert_node(node,cycle_machine(node))
    nxt,res,coepoch = p.run(conf)
    insert_node(node,nxt)
    # Censor the current node from the predicates. If the result is
    # computable then the predicate is already falsified, if not it is
    # confirmed.
    return (nxt,res,coepoch{predicates=coepoch.predicates.delete(node)})
```

Listing 5.38: The algorithm for creating predicates.

At the end of the computation Antisthenis goes through all the non-computable predicates and asserts that they are indeed non-computable. The ones that turn out not to be non-computable are collected and set as *copredicates* in the epoch. The node is then reevaluated with the new epoch. Due to the incremental nature of Antisthenis only the parts of the graph that are dependent on the predicates are reevaluated. The process is described in listing 5.39

```
def safe_run(n,conf):
    # To check if new copredicates were added
    initial_copred_num = len(conf.copredicates)
    # Run the process
    coepoch,res,coit_p = lookup_node(n).run(conf)
    # Remember: machines censor their own reference from the predicate
    # and no predicates.
    assert n not in coepoch.predicates
    assert len(intersection(conf.epoch.predicates,coepoch.predicates)) == 0
    # For efficiency any result we find we register all resutls we
    # find that may be encountered later as predicates.
    conf.copredicates.push(n)
    # For each non computable predicate
    for noncomp_node in coepoch.predicates:
        # Run the predicate to check if indeed it is non-computable
        res = safe_run(lookup_node(noncomp_node), conf)
        if is_computable(res):
            # If it is actually computable then register it a copredicate.
            conf.copredicates.push(noncomp_node)

    if len(conf.copredicates) > initial_copred_num:
        return lookup_node(n).run(conf)
    else:
        return res
```

Listing 5.39: Algorithm for handling the predicates.

Another way to look at predicates is as deferred parts of the coepoch, to get the full coepoch we also need the coepochs of the predicates. The copredicates then are a forcing mechanism where Antisthenis assumes that some of deferred parts are actually invalid. Antisthenis is lazy with these coepochs first and foremost because it depends on it to handle cycles but also because coepochs may be inhibited by absorbing elements, and it may not need to evaluate them at all. But if a predicate survives both the process stack based censoring that we just described and being inhibited by the operators, Antisthenis has another trick up his sleeve: because it has a completely relativistic view of values, where they are only valid for particular epochs, the only requirement at every moment is that the final result of the computation be correct. Furthermore, the value depends on the machines in the predicate being $\perp$. Therefore, it is enough to check that they are indeed $\perp$ certifying the predicates. The predicates that are falsified in this way are the ones for which there is no other option but to force their value be taken into account via the epoch's copredicates.

5.5.4 Calculating historical query costs

As we saw in section on the query planner, the planner depends on the cost of past nodes to make decisions about the overall usefulness of a plan. A naive approach to this would be to just use the normal min/sum operation set to calculate historical costs. As explained in more detail in that chapter this approach is problematic. We need a way of convincing Antisthenis to "look behind" those materialized nodes a) without completely ignoring the materialized nodes and b) by handling referential cycles more gracefully than declaring nodes non-computable. For this reason we develop the *stochastic costs model*.

The correct way of calculating the expected cost of a node in the future, which we very informally and heuristically attempt to approximate, would be to instead of considering the cost of materialized nodes to be zero, to calculate the likelihood that a materialized node will still be materialized when we encounter it again. This is related not only to an estimation of how many page-writes separate the moment of cost estimation and the actual plan the cost of which is being estimated, but also all the decisions that the garbage collector will make in that time. After FluiDB's budget is exhausted for the first time, for every page write there a page needs to be garbage collected. We considered a couple of options for stochastically modelling the behavior of the GC like assuming that it chooses random pages or random tables, but we could find none that was useful and computationally viable.

For this reason we decided to follow a pragmatic approach to the problem and simply assume that for every materialized node there is a constant probability that it will not still be materialized when we need it, scaling the cost by that factor to get the stochastic cost. Furthermore, under this regime, Antisthenis would encounter many more cycles, rendering virtually all machines non-computable. For that reason we make all values semi computable by introducing a $[0, 1]$ scale where 0 would mean that the value is fully computable and 1 would mean that the value is completely non-computable.

Fortunately this change does not require a complete overhaul of the operator implementations, only the definition of more complex types for cap, bound and result values. Particularly, we qualify each value with two numbers:

- the materialized nodes encountered
- a quantification of the non-computability of the value

Some care needs to be taken so that the values remain independent of the root of the computation.

The cap (listing 5.40) is only qualified by the number of machines corresponding to materialized nodes that are currently in the computation stack, i.e. machines that would have otherwise yielded zero cost.

```haskell
data HistCap a =
  HistCap
  { hcMatsEncountered :: Int
   ,hcValCap :: Min a
   ,hcNonCompTolerance :: Double
  }
```

Listing 5.40: Definition of the type used for capping the cost of historical queries.

The bounds and results are augmented by two new dimensions (listing 5.41):

- The $[0,1]$ computability metric
- The maximum chain of materialized nodes encountered.

And the comparison between bounds and caps is presented in listing 5.42.

```haskell
data HistRes a =
  HistRes
  { hrComp :: Double  -- [0,1] computability metric.
   ,hrMaxMatTrail :: Int -- Maximum trail of materialized nodes
                         -- encountered.
   ,hrRes :: a
  }
```

Listing 5.41: Definition of the type used for both bounds and results of historical queries.

```haskell
-- | We prefer "more computable" results than "less computable" so we
-- scale the actual result by the computability. We also define a
-- computability threshold under which we consinder the value always
-- >.
compare :: HistRes a → HistRes a → Comparison
compare (HistRes c1 t1 r1) (HistRes c2 t2 r2) =
  if
    | c1 > thr && c2 > thr → compare (scale c1 r1) (scale c2 r2)
    | c1 > thr             → GT
    | c2 > thr             → LT
    | otherwise            → compare a b
  where
    thr = 0.7

-- There are no circumstances under which we are going to choose a
-- value that has come through more materialized nodes. The scaling
-- will have damped it enough.
instance Ord a ⇒ Ord (Cert a) where
  compare (Cert _ a') (Cert _ b') = compare a' b'

extExceedsCap _ HistCap {..} HistRes {..} =
  maybe False (hrRes >) (unMin hcValCap)
   || hrMaxMatTrail bnd > (maxMatTrail - hcMatsEncountered)
   || hrComp bnd > hcNonCompTolerance
  where
    maxMatTrail = 5 -- actually this is part of a global config.
```

Listing 5.42: Comparison between bounds and between bound and cap are different. Between bounds we need to account for the computability metric. The cap on the other hand defines a three-dimensional bound () that the bound must fall within in order to not exceed it.

5.6 Conclusion

In this chapter we described in detail the incremental evaluation system we developed for FluiDB that we call Antisthenis. The main goals of Antisthenis are to compute incrementally, to gracefully handle computational cycles and to order the operations in an operator specific way such that the overall computation finishes as soon as possible.

There are two major shortcomongs of Antisthenis:

- Extending it to new operators and even using the existing operators involves a lot of boilerplate. This is an immediate effect of the current version of Antishtenis being highly specific to the FluiDB planer.
- Our current implementation relies heavily on the higher order functional capabilities of haskell that are efficient for what they are, but they stress the garbage collector and are hard for GHC to optimize.

Both of these could be mitigated in a future version of Antisthenis that would be detached from FluiDB and that would replace the Antisthenis process with a more "traditional" data structure.

Further work could also be done to parallelize Antisthenis. Some operations like `min` rely on the operands being evaluated serially to take advantage of early stopping but addition is much more flexible with the order of evaluation. It is clear that parallelism would also be implemented in an operator specific way.

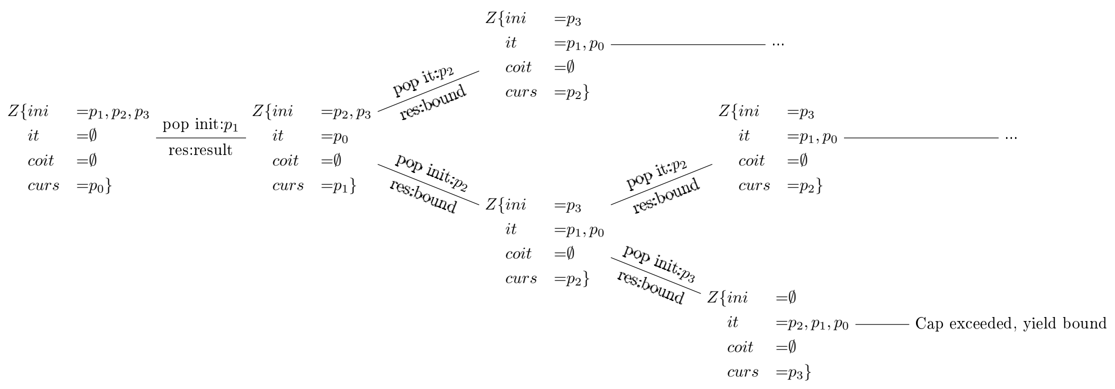
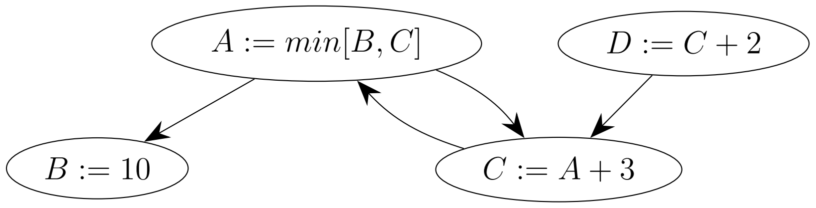


Figure 5.3: A tree of different evolution paths a zipper may undergo during between the calling of a process and the accumulated bound exceeding the cap. p_0, p_4, p_3 and p_4 are subprocesses. The values returned, the specific iteration subset data structure are not presented for brevity and to shift focus to the movement of the subprocesses within the zipper. Nodes in this tree that have two children correspond to the **`ItInit`** constructor, nodes with one node correspond to the **`CmdIt`** or **`CmdInit`** constructor depending on which process set the process in the nex cursor is drawn from. When a concrete value is reached or the cap is exceeded the process stops and the free monad that wraps **`Cmds`** takes the value **`Pure`**.



(a) A recursive cost graph.

$$\left.\begin{array}{l} comp[A]: \\ \ comp[B] \Rightarrow 10 \\ \ \left.\begin{array}{l} comp[C]: \\ \ comp[A] \Rightarrow \bot \end{array}\right] \Rightarrow \bot \end{array}\right] \Rightarrow \bot$$

(b) Computation for A

$$\left.\begin{array}{l} comp[D]: \\ \ comp[C] \Rightarrow \bot \end{array}\right] \Rightarrow \bot pp$$

(c) Naive computation for D

$$\left.\begin{array}{l} comp[D]: \\ \ \left.\begin{array}{l} comp[C]: \\ \ \left.\begin{array}{l} comp[A]: \\ \ comp[B] \Rightarrow 10 \\ \ comp[C] \Rightarrow \bot \end{array}\right] \Rightarrow 10 \end{array}\right] \Rightarrow 10 \end{array}\right] \Rightarrow 10$$

(d) Computation for D

Figure 5.4: Node C is recomputed because the $\bot$ value is predicated on A being in the computation stack.

CHAPTER 6

Execution engine

Don't hit hard the beads, work
is what makes a man.

(Vamvakaris)

Chapter summary

- Tables are stored in a memory file systems and records are stored as POD binary objects organized in 4K pages.
- The extent of each relation involved in a plan is converted to C++ struct and each predicate is converted to a C++ callable class.
- These parameterize the templatized operators to allow the compiler to generate highly specifialized code.
- (TODO) We use an approach reminise of scheme's macro hygene to manage the uniqueness of symbols in the generated code.
- In this chapter we discuss the implementation important operations and their reverse.

The logical plan is translated into a physical plan that has the form of a c++ program. The motivation for using code generation relates to expression level optimizations so it's less work on some fronts (although more work in others). Also gdb for debugging. Remember that compilation time is not pure overhead, we would have to duplicate at least some low level optimizations in the haskell.

6.1 Codegen introduction

Code generation is becoming a more and more common in RDBMSs. Used mainly in in-memory databases, where disk IO does not dominate the runtime, it aims to minimize the overhead of data access function calls, to optimize the operdicates and numerical expressions and to avoid indirection in tight loops. Approaches to code generation fall generally on a spectrum between two extremes:

- Transpilation of the physical plan to a low level programming language like C or C++ for every query
- JIT compilation of small parts of the plan as the query executes.

FluiDB follows the approach of [30] falling far to the former pole of the spectrum. We generate very specific, template-heavy C++ code for every query and we call to an off-the-shelf compiler to generate highly optimized machine code.

We opted for delegating the task to the OS and use use use the tmpfs as a storage layer to our database. The tmpfs filesystem depends on the the shmem module (as of linux v5.13) for handling file operations. shmem is a resizable virtual memory filesystem for linux. Where a typical persistent filesystem stores files in a block device and caches pages in memory for efficiency, shmem keeps files exclusively as pages in the page cache. The OS tries to keep all pages in memory but when resources start running out it writes pages in swap.

Assuming that the pages are not in swap, normal reads for shmemfs using `read()` are equivalent to copying pages from the page cache to the user space. `shmem` writes on the other hand operate directly to the pages. We can mitigate the copying overhead of reads using `mmap` which will remap the page to the address space of the application.

The problem with this approach is that, while it allows us to minimize copying, we still need to run system calls in tight loops, which can be very computationally expensive.

This can't be completely mitigated unless we move the "storage" layer to the userspace, re-implementing the memory management that we get for "free", in terms of engineering effort, from the shmem module. Another problem with the latter is that, as things stand in FluiDB, the code generated has the file names that correspond to materialized relations hardcoded. An approach completely dependent on malloc/free for memory management would complicate the access to the materialized relations significantly.

On the other hand this approach allows FluiDB to easily be adapted to operate over any filesystem backed by different storage technologies like non-volatile memories.

6.2 Data layout

Before we get into the details of the actual physical plan (in the form of C++ code) we need to go into some assumptions about the layout of the code that is assumed by the primary data.

FluiDB is a *row store system* and it depends on the filesystem for itnermediate result lookup and page management. For code generation to make sense the file system is, in particular, a tmpfs filesistem that resides entirely in ram. This is not an ideal solution performance-wise and in the future we plan on using a less OS-reliant way of managing storage resources but it is good enough for now. In particular we use one file per table or intermediate result, and each file is simply a sequence of recods organized into pages. For now FluiDB does not support any kind of indexing or compression. FluiDB can, of course, be run over any filesystem but non-memory based file systems diminish benefit of code generation.

With this in mind there are three parts to understanding the principles of FluiDB storage:

- The format in which primary data is inserted into the database.
- The layout of the data withing the database
- The transformation from the former to the latter.

6.2.1 Initial data conversion

Starting with the initial data, as it stands, for FluiDB to be adapted to a particular dataset it requires the primary data in CSV format and some haskell code describing the shape of the data and the database configuration. Our experiments so far have been revolving around the SSB TPC-H benchmark so the format expected is the plaintext format that dbgen [79] generates. This is comprized by two steps: first a haskell program parses the CSV records into standard-layout binary objects that can be directly cast to C/C++ standard-layout structs. These binary objects are stored one after the other in a flat binday file with the extension `.bama`, refering th the *BAMA* library that FluiDB generated calls into to execute operators. The end result of what we want to do to be able to execute code similar to the one presented in listing 6.1.

```cpp
template<typename R, size_t batch_size=2000>
void bama_to_dat(const std::string& bama_file, const std::string& dat_file) {
  int fd;
  size_t read_bytes;
  std::array<R, batch_size> batch;
  // Open the file in binary mode
  fd = ::open(bama_file.c_str(), O_RDONLY);
  // The writer is the object used by all the libraries and it will
  // write each record in pages.
  Writer<R> w(dat_file);
  do {
    // Read a batch of data,
    read_bytes = ::read(fd, batch.data(), sizeof(batch));
    // Use the writer object to write each record.
    for (size_t i = 0; i < read_bytes / sizeof(R); i++) {
      w.write(batch[i]);
    }
    // Keep reading until a batch is cut short.
  } while (read_bytes == sizeof(batch));
  // Wrap up.
  w.close();
  close(fd);
}
```

Listing 6.1: For standard FFI communication C++ structs that do not contain fancy constructors

For this to work we refer to the C++ standard [70]. The types that represent table rows must be standard layout. According to the standard:

> A class S is a standard-layout class if it:
>
> - has no non-static data members of type non-standard-layout class (or array of such types) or reference,
> - has no virtual functions and no virtual base classes
> - has the same access control for all non-static data members,
> - has no non-standard-layout base classes,
> - has at most one base class subobject of any given type,
> - has all non-static data members and bit-fields in the class and its base classes first declared in the same class, and
> - has no element of the set M(S) of types as a base class, where for any type X, M(X) is defined as follows. [Note: M(X) is the set of the types of all non-base-class subobjects that may be at a zero offset in X.]
>   - If X is a non-union class type with no (possibly inherited) non-static data members, the set M(X) is empty.
>   - If X is a non-union class type with a non-static data member of type X_0 that is either of zero size or is the first non-static data member of X (where said member may be an anonymous union), the set M(X) consists of X_0 and the elements of M(X_0).
>   - If X is a union type, the set M(X) is the union of all M(Ui) and the set containing all U_i , where each U_i is the type of the i th non-static data member of X.
>   - If X is an array type with element type X_e, the set M(X) consists of X_e and the elements of M(Xe).
>   - If X is a non-class, non-array type, the set M(X) is empty.

The records we generate certainly conform to these requirements. An example is the supplier row in 6.2. Therefore the object is trivially copyable and occupies contiguous bytes of storage. This means that we can safely write each record `R` as `sizeof(R)` contiguous binary data to a file and expect to find the same value of `R` when we read it. Based on this we can safely copy binary record objects from memory to disc and visa versa.

```
class Record {
public:
  Record(unsigned __s__suppkey, fluidb_string<25> __s__name,
         fluidb_string<40> __s__address, fluidb_string<16> __s__city,
         fluidb_string<16> __s__nation, fluidb_string<13> __s__region,
         fluidb_string<15> __s__phone)
    : s__suppkey(__s__suppkey),
      s__name(__s__name),
      s__address(__s__address),
      s__city(__s__city),
      s__nation(__s__nation),
      s__region(__s__region),
      s__phone(__s__phone) {}
  Record() {}
  std::string show() const {
    std::stringstream o;
    o << s__suppkey << " | " << arrToString(s__name) << " | "
      << arrToString(s__address) << " | " << arrToString(s__city) << " | "
      << arrToString(s__nation) << " | " << arrToString(s__region) << " | "
      << arrToString(s__phone);
    return o.str();
  }
  bool operator==(const Record& otherRec) const {
    // compare each field ...
  }
  bool operator!=(const Record& otherRec) const {
    // compare each field ...
  }
  unsigned s__suppkey;
  fluidb_string<25> s__name;
  fluidb_string<40> s__address;
  fluidb_string<16> s__city;
  fluidb_string<16> s__nation;
  fluidb_string<13> s__region;
  fluidb_string<15> s__phone;
};
```

Listing 6.2: The supplier row representation in the generated C++ code. The fluidb_string type is a constant size arraw of characters.

However, in the case of parsing we need to take great care with byte alignment which is compiler dependent. Fortunately Clang and GCC informally aggree on the algorithm for `alignof(<cls>, <member>)` for standard layout objects. The algorithm is presented in 6.3

TODO: translate the figure to python or sth more understandable.

```
schemaPostPaddings :: [CppType] → Maybe [Int]
schemaPostPaddings [] = Just []
schemaPostPaddings [_] = Just [0]
schemaPostPaddings schema = do
  elemSizes ← sequenceA [cppTypeSize t | t ← schema]
  spaceAligns' ← sequenceA [cppTypeAlignment t | t ← schema]
  let (_:spaceAligns) = spaceAligns' ++ [maximum spaceAligns']
  let offsets = 0 : zipWith3 getOffset spaceAligns offsets elemSizes
  return $ zipWith (-) (zipWith (-) (tail offsets) offsets) elemSizes
  where
    getOffset nextAlig off size =
      (size + off)
      + ((nextAlig - ((size + off) `mod` nextAlig)) `mod` nextAlig)
```

Listing 6.3: Algorithm to infer the padding of members according to the Itanium ABI.

Once the bama files are generated C++ code is generated for parsing the file calls into the C++ function `bama\_to\_dat` that is parametric to the type of the object being and uses the BAMA library to write objects, thus making sure that the data is readable by the operators. `bama\_to\_dat` simply reads the input `.bama` file as a stram of records of size `sizeof(Record)` casting the bytes with `reinterpret\_cast<Record*>` into the record. It then use the bame record writing facility `Writer<R>::write` that takes care of organizing the record into pages. Thus the final *data file* is created that is ready for use by the generated code.

```
#include <bamify.hh>
class Record {
  // ...
};
int main(int argc, char* argv[]) {
  bama_to_dat<Record>("supplier.bama","supplier.dat");
}
```

Listing 6.4: Convert a bama file to a paged data file.

6.2.2 Pages

The `.dat` files are not very complicated. The basic block of the file is the `Page`, and the file is simply a raw squence of pages. A page is constant-size data structure that contains up to $\left\lfloor \frac{S_{rec}}{S_{pg}} \right\rfloor$ *whole* records where S_{rec} is `sizeof(Record)` and S_{pg} is the size of the page. All pages must contain as many whole records as can fit except the last one.

All transactions with the storage are made at the page level: we either read or write only entire pages. These operations are abstracted by the `Reader` and `Writer` classes. We use one more level of abstraction for convenience, the higher order `eachRecord` function. To demonstrate what the interface looks like we present the implementation of `eachRecord` 6.5.

```
// Fn could be instantiated to std::function<void(const R&)> but that
// will *always* forbid f from being inlined.
template <typename R,typename Fn>
inline void eachRecord(const std::string& inpFile,Fn f) {
  Reader<R>' reader;
  size_t i = 0;
  reader.open(inpFile);
  while (reader.hasNext()) {
    i++;
    f(reader.nextRecord());
  }
  reader.close();
}
```

Listing 6.5

As alluded to in the previous section both the `Writer` and `Reader` use `reinterpret\_cast` to "serialize" and "deserialize" the data respectively.

6.2.3 In-place sorting

It is important for the planner to have full control of the storage budget and assume that no significant memory is required for evaluating each operator. For that reason we require that all operators' algorithms have constant space complexity. This may mean major compromizes for some algorithms with respect to the time complexity like join and aggregation. Fortunately we can mitigate that by taking advantage of the set semantics of FluiDB relational algebra and sorting input tables in-place before running joins or aggregations. Our particular implementation of in-place sorting hinges on the `RecordMap` type that provides C++ random access iterators iterators to the records of a file, abstracting the page reads and writes. We pass these iterators to `std::sort` (see listing 6.6) which runs insertion sort for small ranges to take advantage of the processors reorder buffer and quicksort for longer ones.

```cpp
RecordMap<size_t> fs("/tmp/removeme.dat");
std::sort(fs.begin(), fs.end());
```

Listing 6.6: Using a `RecordMap` to sort the records of a file.

The operation of `RecordMap` is very simple. Each iterator is paired with the page that contains the record it points to, when an iterator is the only iterator pointing inside a page and is incremented or decremented and no longer points to the same page the old page is written back to the file and the new page is read. When the only iterator pointing to a page is deleted the page is written to the file. More than one iterators pointing to the same page do not maintain different copies of that page and the last one to be destroyed or to leave the page triggers the page to be written back to the file.

6.3 Physical planning

The fundamentall logic of the code generator is fairly simple: The input of the code generator is a logical plan generated by the FluiDB planner in the form of a sequence of transitions. The reader is reminded that each transition has one of three kinds:

- Trigger of a t-node. The input n-nodes at the time of the trigger are materialized and the trigger itself materializes a subset of the output n-nodes
- Reverse trigger of a t-node, which represents the materialization of a subset of the input n-nodes from materialized output nodes
- The deletion of an n-node.

Transitions are meant to be executed in the order they appear in the received sequence to preserve correctness. However there is no one-to-one correspondence between operators and transitions. Instead, as we saw in the discussion about the e , each operation corresponds to a cluster of connected t-nodes. The first step of the code generator, therefore, is to group the low level transitions received by the planner into higher level batches that correspond to exactly one operation. This process is driven by intermediate n-nodes, ie helper nodes that do correspond to materializable relations, but are rather part of the graph to reify the valid combinations that the planner is allowed to materialize. The constraint we are trying to preserve is batch the low level transitions in such a way that the high level transition does not materialize any intermediate nodes.

Since intermediate nodes are always internal to clusters and no cluster shares a t-node with another cluster, each batch of transitions corresponds to exactly one cluster, except for deletion transitions which are standalone. Furthermore, each cluster corresponds to exactly one relational operator, which we also include in the higher order transition.

The C++ AST is expressed as a tree of haskell algebraic data types. They do not capture the entire C++ language, only the following concepts concepts:

- Function declaration
- Function application and arguments
- Expressions
- Code symbols
- Assignment
- Includes

- Literals
- Classes
- Member clarations (for record printing)
- Algorithm selection
- Global declarations

The code generated version of each operator is parameterized by the record types of its inputs and outputs and highly specific code is generated by the Cq++ templating system. Operators are reified as C++ code within the context of a state monad we call CodeBuilderT that generates C++ struct for each record type encountered and registers the types of the argument and return types for declaration.

Therefore, an important aspect of code generation is bridging the gap emerging from the logical plan having a form fundamentally different to the form of a C++ program. In the case of C++, the logic and type declarations reside in distinct parts of the code. The logical plans emitted by the planner, however, are a sequence of operations on data where the types, which for our purposes correspond to the table and subexpression schemata, are implicit in the logical plan.

`CodeBuilderT` is then equipped with operations to insert unique declarations of functions and structs.

6.3.1 AST to literal code

In fludb the C++ AST is converted to a literal code string. This is done in two steps:

- Translation of the AST into fragments of soft literal code, that is code in a format similar to that of a string, which has the property to maintain the uniqueness of symbols under concatenation.
- Concatenation of soft code fragments and translation to a literal string representing the C++ code.

This is done by the `Codegen` typeclass. Any type implementing Codegen can be transformed to an IsCode type. IsCode is any type that can be directly converted into a string of C++ code. For now consider that IsCode only refers to normal strings. In this section we will look at the transformation of a valid AST into code.

Code hygene

A major concern in any transpiler, or code generation system generally, is the generation of unique symbols. The problem is similar to the one regarding scheme's hygienic macros, although the problems for those are much more complex than our particular case. This is mostly because our plans expand to a different language and expansion happens exactly once. Consider the following C++ AST that we want to transform into code.

$$R_1 \bowtie_\theta R_2$$

The most naive and straightforward way is to convert it directly into a C++ string.

```
struct R1 { … };
struct R2 { … };

bool theta (R1 r1, R2 r2) {
  …
}

int main () {
  …
  auto op = Join<R1,R2>(theta,"R1.dat", "R2.dat");
  op.run();
}
```

This approach works for a very narrow class of cases where the names `theta`, `R1`, and `R2` do not collide with any other names in the program. The obvious approach would be the Common Lisp approach to "hygene" where we generate a new symbol for each identifier. The naive and obviously wrong way to do that would be

```
struct R1_1 { … };
struct R2_2 { … };

bool theta_3 (R1_4 r1, R2_5 r2) {
  …
}

```

```
int main () {
  ...
  auto op = Join<R1_6,R2_7>(theta_8,"R1.dat", "R2.dat");
  op.run();
}
```

This goes too far and discriminates between names that we need to match.

```
-- todo
```

Listing 6.7: Consistently unique symbols

New problems arise if we want to concatenate different code snippets. Consider thecase

```
-- todo
```

Listing 6.8: Variable reuse

If these generated code snippets uniquify their variables separately the variable name a will be re-declared making the compiler unhappy. We resolved this issue by introducing the SoftUNameCode functor. SoftUNameCode a implements the IsCode interface as long as a implements it as well.

```
data SoftUNameCode a = SoftUNameCode {
  uCodeTail :: Maybe (UniqueSymbol,SoftUNameCode a),
  uCodeHead :: a,
  uCodeMaxId :: UId
  }

data UniueSymbol a = UniqueSymbol {
  symbolUId :: UId,
  symbolMkName :: UId → a
  }
```

Listing 6.9: Variable reuse

What this amounts to is a list of code objects interleaved with unique symbol names. Each unique symbol name is a combination of an unique id and a function that would generate a unique string of code based on that id. This unique id must be incrementable.

When two `SoftUNameCode` objects are concatenated, in the simple case we aim for the variable names to be completely disjoint. This is achieved by simply incrementing all symbolUIds in the right hand side of the concatenation operator by the left hand side uCodeMaxId.

In the slightly more complex case where we know that the same unique symbol is both on the left and the right hand side of the concatenation operator we use co-join `SoftUNameCode` a into `SoftUNameCode (SoftUNameCode a)` pushing the shared symbol in the internal layer. When concatenating SoftUNameCode, only the top layer of the right hand side is changed. After concatenation the functor is flattened:

With this method we can flexibly concatenate code keeping certain variables hygienic while maintaining reference to others.

6.4 Operator implementations

Now that we have an idea of how the code is actually generated. All operator implementations live in the BAMA library which branched out from [30]. The BAMA library makes heavy use of templates and constexpr to generate very query-specific assembly.

6.4.1 Generated code structure

The FluiDB code generator generates a couple of different kinds of C++ components to construct the final file. In this section we will discuss the kinds of C++ structs that FluiDB generates in order to parameterize the BAMA operators at compile time, leveraging the C++ template system.

In the following subsections when we say *compile time* we refer ti the compile time of the generated code, not compile time of FluiDB.

Maybe types

Maybe types are used to indicate optional outputs of operators. For example an operator σ_p may yield $\sigma_p A$ or $\sigma_{\neg p} A$ or both. On one hand the precise outputs that are required are important for performance and memory budget management, and on the other any combination of these outputs can and should be generated through one pass over the input. Furthermore information about which outputs are required is know at compile time so the generated code should be specific to the combination of outputs that are required.

To address this, instead of passing in simple `std::string` as paths, we pass one of two types `Nothing` or `Jusr` (listing 6.10) that wrap the file path. Both these types contain an `isNothing` static constexpr type that can be used by `if constexpr (...)` expressions in the selection operator to help the compiler generate highly specialized machine code.

```cpp
template <typename T>
struct Just {
  Just(T t) : value(t) {}
  T value;
  constexpr bool operator==(Just<T> const& j) const {
    return this->value == j.value;
  }
  static constexpr bool isNothing = false;
};
template <typename T=std::string>
struct Nothing {
  Nothing(T s) {}
  Nothing() {}
  static constexpr bool isNothing = true;
};
```

Listing 6.10: The type level maybe

Record types

FluiDB is a row store engine and handles table rows by generating record types specific to each table, where objects of that type are rows of the corresponding table. We went over some aspects of the the genrated record types when discussing the conversion of primary data to FluiDB specific binary data. Records types directly or indirectly parameterize all operators. A selection operator σ_p instance must be parameterized by the record type of its input which will be the same as the output. A projection π is parameterized by the records of both the input and the output type. A generated record type must must have some structural properties to function properly with BAMA and the rest of the generated types:

- First and foremost it must be standard-layout type provide so it can be trivially copied from and to binary blob files. This means that there should be no virtual members, the destructor must be trivial and no static members. We further impose the constraint that they must not have a base class to avoid some more intricate constraints imposed by the C++ standard on standard-layout classes.

- it must provide non-uniquified names for each field of the row it represents so other generated types (like predicates and subtuple extraction types) have direct access to the underlying data.

- They should be comparable by equality ($=$), inequality ($\neq$) and they should be orderable $<$. The actual semantics of the ordering are not important as long as there is a deterministic way of ordering records of the same type.

- They should be hashable. We get that for free since they are standard layout objects but it is an important property that is useful for equijoins as we will see in the section about equi-joins.

- Finally every record type must implement an `std::string show()\textasciitilde {}` function that serialized the contents into a human readable string.

Predicate types

Predicate types have the function of n-ary boolean functions $A_1 \times ... \times A_k \rightarrow \{T, F\}$ useful for selections and θ-joins so in practice they are unary or binary. They are passed as template arguments to the operators so the compiler has the chance to inline them to avoid a function call. To avoid too much code clutter they are expected to provide the types of the function domain (demonstrated in listing 6.11) in order to minimize the number of template parameters and keep the generated code relatively human readable.

```
class Predicate3421 {
  typedef Record123 Domain0;
  typedef Record32 Domain1;
  bool operator() (const Domain0& rl, const Domain1& rr) {
    return rl.__field1 < rr.__field2;
  }
}
```

Listing 6.11: The shape of a generated predicate type.

Record transformation types

Record transformations, much like predicates are callables representing pure functions, only the codomain is a record type, instead of a boolean. They are used by joins to combine the matching records, by projections and aggregations to produce new records from the inputs, by equi-joins to extract the subtuples to be checked for equality, etc. Much like predicates they can be queried for their domain and codomain to reduce the number of template arguments in operators (see listing 6.12)

```cpp
class Transform {
  typedef Record123 Domain0;
  typedef Record32 Domain1;
  typedef Record10 Codomain;
  Codomain operator() (const Domain0& l, const Domain1& r) {
    return Record10(l.__key,l.__field1, r.__key, r.__field2);
  }
};
```

Listing 6.12: A record transformation type defines objects with no internal state that are callable.

Operators

Operators are not generated but they are parameterlized by all the kinds of gernerated code we mentioned. They are clases that are constructed using maybe-filenames, record transformers, record types, and predicates and implement the `run()` method which actually runs the internal code. A simple operator is selection. An abbreviated version of it, which highlingts the shape of an operator class is presented in listing 6.13.

```cpp
template <typename Predicate,
          typename PrimaryOutType,   // Maybe(std::string)
          typename SecondaryOutType  // Maybe(std::string)
          >
```

```cpp
class Select {
  typedef typename Predicate::Domain0 Record;

public:
  Select(PrimaryOutType prim, SecondaryOutType sec, std::string in)
    : primary_file(prim), secondary_file(sec), infile(in) {
    static_assert(!PrimaryOutType::isNothing || !SecondaryOutType::isNothing,
                  "Both primary and secondary output files are Nothing.");
  }

  ~Select() {}

  void run() {
    // ...
  }

  void print_output(size_t x) {
    // ...
  }

private:
  PrimaryOutType primary_file;
  SecondaryOutType secondary_file;
  std::string infile;
  static Predicate predicate;
};
```

Listing 6.13: The selection operator. It is parameterized by the predicate and the primary and secondary output types. Enough information about these values is known at compile time such that the compiler can generate highly speclalized code.

Unfortunately, even recent iterations of the C++ standard do not include template type inference for classes and structs, therefore, to make the generated code simpler we wrap the constructor into a function like shown in listing 6.14. This allows the generated code for selection to look like the one presented in listing 6.15.

```cpp
// C++17 can only infer typenames (primaryout secondaryout) in
// function templates.
template<typename Predicate,
         typename PrimaryOutType,   // Maybe(std::string)
         typename SecondaryOutType  // Maybe(std::string)
         >
auto mkSelect (const PrimaryOutType prim,
               const SecondaryOutType sec,
               const std::string& in) {
  return Select<Predicate,
                PrimaryOutType,   // Maybe(std::string)
                SecondaryOutType> // Maybe(std::string)
    (prim, sec, in);
}
```

Listing 6.14: The C++ declaration of the select.

```cpp
int main() {
  // ...
  {
    auto op = mkSelect<Predicate32>(Just("data123.dat"), Just("data53.dat"),
                                    "lineitem.dat");
    op.run();
    op.print_output();
  }
  // ...
  return 0;
}
```

Listing 6.15: A block representing a particular operator.

6.4.2 Select algorithm

The selection algorithm is likely the simplest of the implemented ones because FluiDB has implements no intexes and makes no assumptions about the ordering of the data.

In the forward variety it is implemented as either a partition or a selection depending on which of the outputs it is materializing. In its backward variety it is essentially a union $A \equiv \sigma_p A \cup \sigma_{\neg p} A$. As we discussed already (listing 6.14) bama expects only a predicate class as a template parameter and three filnames as its arguments.

It is important to remind the reader that no order is expected to be preserved by union, which is to say that the transition $A \Rightarrow \{\sigma_p A, \sigma_{\neg p} A\} \Rightarrow A$ does not preserve the order of the nodes. The reader is also reminded that FluiDB does not support `NULL` values to preserve the correctness of the reverse select operation.

6.4.3 Projection algorithm

The implementation of the projection algorithm is parameterized by two template arguments that extract complementary subtuples from an input relation, affording each one with a common unique subtuple to facilitate the inverse operation. The inverse of a projection then is simply an equijoin between the two produced slices based in that shared unique subtuple.

While the BAMA side of the calculation is fairly straigtforward, on the haskell side it is slightly more complex. The first piece of the puzzle is the representation of the projection. When the query is initially processed so that projections are augmented to expose unique subtuple we actually change the represetation of projections from `QProj [(e,Expr e)]` to `QProjI [e] [e] [(e,Expr e)]`. The two extra parameters typed `[e]` represent the complement of the projection (the columns from the input not exported by the projection) and a unique subtuple that the complement and the projection must have in common. `[(e,Expr e)]` is the normal projection. We make this transformation because the easiest and cheapest place to calculate the first and second parameters (complement and correspondence) is during the projection augmentation phase that happens immediately after parsing. Therefore we have all projections carry information about the complement at the level of the operator.

In particular this process is combined with remapping of uniuque subtuples into the projection. For example a projection. During the unique column exposure process the query `select p_color from part` is not valid because `p\_color` is not unique. For that reason, during preprocessing we remap the `p\_partkey` column since `p\_partkey` is the only unique column of `part` in order to satisfy the constraint that every valid relation in FluiDB needs to have at least one subtuple that is unique within that relation.

```sql
select p_partkey,p_color from part
```

In order to be able to reconstruct the input we require that at least one unique subtuple is shared between the primary projection relation and the complement. In the example demonstrated this means the `p\_partkey` column is duplicated in both output tables of the operator. The particular valid projection operator that would be generated the **`QProj`** is presented in listing 6.16.

```haskell
op =
  QProjI
    -- All keys except p_color
    [p_partkey,p_name,p_mfgr,p_category,p_brand1,p_type,p_size,p_container]
    -- The unique subtuple (just one column)
    [p_partkey]
    -- The the actual projection
    [(p_partkey,E0 p_partkey),(p_color,E0 p_color)]
```

Listing 6.16: The projection operator produced from the SQL query **`select`** `p\_partkey,p\_color` **`from`** `part`.

The algorithm for reverse projection is almost a standard join. The only difference is that in our semantics of join, the relation $A \bowtie_{a=b} B$ contains both the column a and the column b, while when joining a projection with it's complement we only create a single copy of the shared subtuple. We will see more details about join in the next section.

6.4.4 The join operator

We distinguish between two kinds of join operators: the equijoin and the general θ-join. The former is specific to cases where subtuples from each table are compared and the implementation is simply a hash join, while the latter is used for general predicates and is essentially a piplelined $\sigma(A \times B)$. The important detail worth noting about join algorithm implementations in FluiDB is that they must be able to produce antijoin relations $A \cancel{\ltimes} B$, $A \cancel{\rtimes} B$ simultaneously with the join relation $A \bowtie B$.

It is important for join to be space efficient, and particularly constant-space, due to the assumption made by the planner that it has control over memory. For general θ-joins we use nested loop joins. For equijoins we take advantage of the assumption that the tables are unordered and use merge join after sorting the input tables in-place. A sketch of the algorithm is presented in listing 6.17

```python
# This is a pythonic way of detecting which input stream finished is.
class LeftFinished:
    pass
class RightFinished:
    pass


def lnext(it):
    """If iteration finishes throw LeftFinished"""
    try:
        return next(x)
    except:
        raise LeftFinished

def rnext(it):
    """If iteration finishes throw RightFinished"""
    try:
        return next(x)
    except:
        raise RightFinished

@template(extractl,extractr,combine)
def merge_equijoin(in_l,in_r,anti_l,out,anti_r):
    # In place sort each of the inputs
    sort(in_l,key=extractl)
    sort(out_l,key=extractr)
    # Define iterators and current values for each of the inputs.
    it_l = iter(in_l)
    val_l = next(it_l)
    it_r = iter(in_r)
    val_r = next(it_r)

    try:
        # The actual merge join algorithm
        while True:
            # Gather blocks of equal records from left and right and write
            # out their product.
            if extractl(val_l) == extractr(val_r):
                # Gather a sequence of equal-key values from the left
                val_ls = []
                tmp = val_l

                while extractl(val_l) == extractl(tmp):
                    val_ls.append(val_l)
                    val_l = lnext(it_l,None)

                # Gather a sequence of equal-key values from the right
                tmp = val_r
                val_rs = []
                while extractl(val_r) == extractl(tmp):
                    val_rs.append(val_r)
                    val_r = rnext(it_r)

                # Write out their product.
                for l,r in product(val_ls,val_rs):
                    out.write(combine(l,r))

            # Push the non-equal records to the antijoins
            while extractl(val_l) < extractr(val_r):
                anti_l.write(val_r)
                val_r = rnext(it_r)
            while extractl(val_r) < extractr(val_l):
                anti_r.write(val_l)
                val_l = lnext(it_l)
    except LeftFinished:
        for l in it_l:
            left_anti.write(r)
    except RightFinished:
        for r in it_r:
            right_anti.write(r)
```

Listing 6.17: The equi-join algorithm first sorts in place the inputs w.r.t. the equal subtuples and then merges them.

The type of the join operator is defined in terms of a template such that it can be specialized at compile time (see listing 6.18). The template arguments are the following:

- Three file paths are provided, `OutFile`, `LeftAnti` and `RightAnti` are each either of type `Nothing` indicating to the compiler not to generate any code that relates to the particular output or of type `Just<std::string>` indicating to the compiler the opposite.
- The subtuple extraction functions `LeftExtract` and `RightExtract`. These are record transformation functions that extract the subtuple that needs to

```
template <typename LeftExtract, typename RightExtract, typename Combine,
          typename OutFile,    // Maybe(std::string)
          typename LeftAnti,   // Maybe(std::string)
          typename RightAnti>  // Maybe(std::string)
class Join;
```

Listing 6.18: Class declaration of the join operator

The reverse join operator is composed of two pipelined steps:

- A projection on the join output to get a semijoin
- A union of the semijoin with the antijoin to get the input table.

As with the selection operator, the correctness of this reverse operation is predicated on the fact that FluiDB has no notion of **NULL** values.

6.4.5 Aggregation and sort algorithms

The main challenge with implementing aggregations in the context of FluiDB is maintaining constant space during its evaluation, as the planner tries to use as much of the memory budget as possible to maintain intermediate results. Therefore we opted against using an auxiliary hash map used to group records. Like we did to implement joins we perform aggregation in two steps: first we sort the records in place based on the grouping columns and then aggregate in a single pass.

CHAPTER 7

Evaluation

Something about what is good.

(Zizek)

Chapter summary

- FluiDB is well suited for join-heavy star schemata so we evaluated using the SSB-TPC-H benchmark.
- Our evaluation shows that FluiDB is able to plan around the space constraints and come up with plans that materialize intermediate results that are useful for future queries.
- FluiDB is generally faster than the baseline due to caching but at times may be slower when it receives "unexpected" queries.
- FluiDB generally performs better when allowed larger memory budgets but this speedup is based on heuristic assumptiosn that sometimes break in interesting ways.

We based our evaluation on the Star Schema Benchmarkx (SSB) [48] which is a variation of TPC-H. Like TPC-H it models the data warehouse of a wholesale supplier. It is organized in four "flights" (groups):

- The first flight performs a single join between the Lineorder table and the small Date dimension table.
- Queries in Flight 2 join the Lineorder, Date, Part and Supplier tables. They aggregate and sort data on Year and Brand.
- Starting from a specific customer, supplier, region and date, flight 3 selects into specific cities and years.
- Flight 4 is the most complex one, as it joins all tables.

Each flight contains 3 queries making a total of 12 distinct queries.

As the name suggests these queries query a star schama. The star schema is derived from the TPC-H schema (figure 7.2) by merging the `linenumber`, `orders` and `partsupp` and completely dropped tables `nation` and `region` tables as shown in figure **??**.

The particular queries contained in the workload are presented in listing [ref] in the appendix (**??**).

We generated a workload where queries were compiled and planned one by one in a loop for and run them over a database of data generated with a modified version of the TPC-H dbgen [80] with scaling of size 1, meaning that the total primary data is around 1GB in CSV format. The exact size of the data generated after formatting them to the format required by FluiDB (described in chapter **??**) are

```
$ du -sh *.dat
4.1M    customer.dat
256K    date.dat
757M    lineorder.dat
31M     part.dat
268K    supplier.dat
```

Which makes a total of approximately 200k pages.

The lowset budget within which FluiDB was able to plan all 12 queries of SSB-TPC-H was 2200k pages.

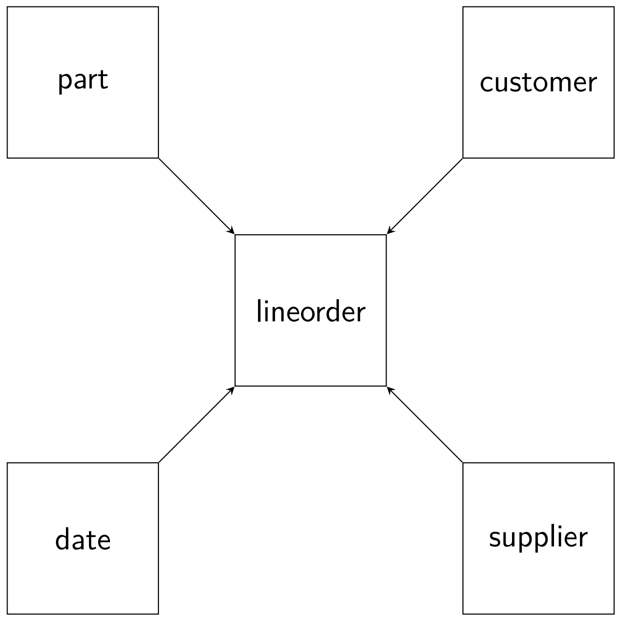


Figure 7.1: The foreign key links in a SSB-TPC-H

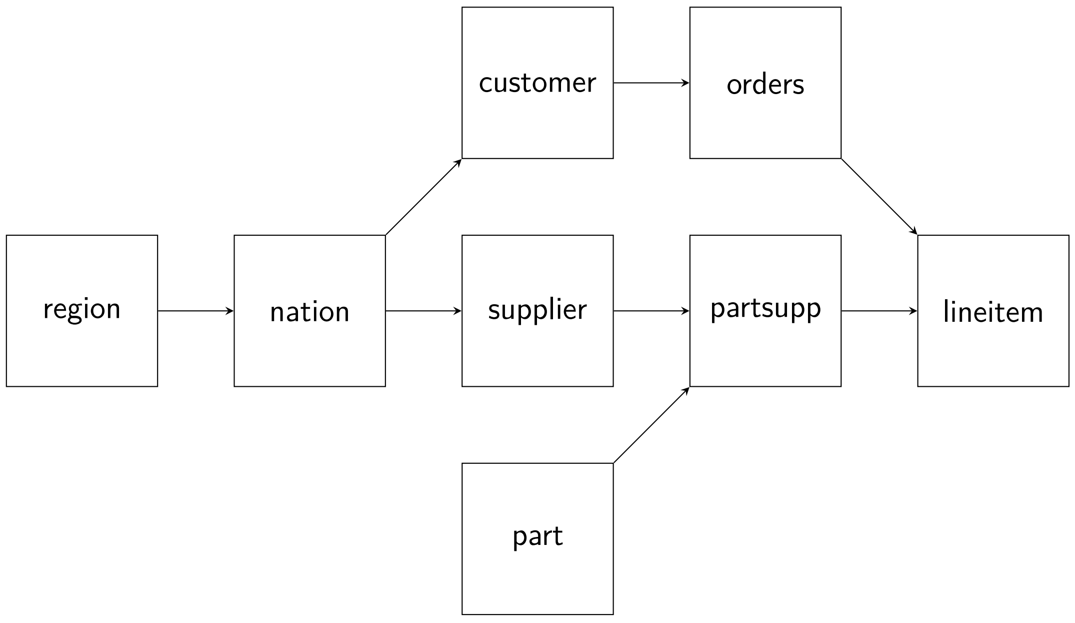


Figure 7.2: The foreign key links in a traditional TPC-H schema

In our expriment we use page IO as a proxy for performance, despite the fact that FluiDB is an in-memory database. We think that this is a reasonable experimental approach because, as FluiDB leans heavily on code generation, it is unlikely that the actual intruction retiring will have a major impact on the perofrmance. Instead the perfomance cost is dominated by page IO that will certainly cause cache misses at the LLC. Another thing to note is that FluiDB generates code that focuses on performance and not on compilation time, making heavy use of metaprogramming like **`constexpr`** and temlpates. This makes for fairly slow compilation times (approximately 1 sec to 2 sec with `-O2`). There are ways to speed up compilation time like using pre-compiling header files [81] and fine-tuning the compiler optimization passes. These techniques are beyond the scope of this work, FluiDB is focused on analytics workloads that do not include sub-second queries.

It is clear from figure 7.3 that even at the minimum budget FluiDB is able to come up with an efficient plan at every case. In some cases, however the garbage collector is forced to delete tables that need to be recreated causing FluiDB to be sporadically less performant than the base case.

For a larger budget the FluiDB is able to store more useful intermediate results as demonstrated in figure 7.4

An interesting point here is the plan for evaluating query 8 is more expensive in the workload run under laxer budgetary constraints. This is strange

The top level explanation is that the lax-budget plan needs to evaluate $lineorder$ via the reverse trigger of a join as it was deleted during evaluation of query 7. Mysteriously during the strict-budget planning the $lineorder$ relation is readily available during planning of query 8! The key is slightly earlier in the workload. FluiDB, materializes in query 5 all the join

$$Q := supplier \bowtie_{lo\_suppkey=s\_suppkey} lineorder$$

and corresponding antijoins Q_1 and Q_2 making the node $lineorder)$ deletable. However, when running the workload under strict budgetary constraints, it is forced to garbage collect shortly after materializing said join at a moment while both Q and $lineorder)$ are protected (see section 4.2.2 on garbage collection). Therefore, FluiDB is forced to delete both Q_1 and Q_2. This makes $lineorder)$ non-deletable when planning for query 5 finishes.

On the other hand, whith laxer budgetary constraints, no garbage collection is triggered

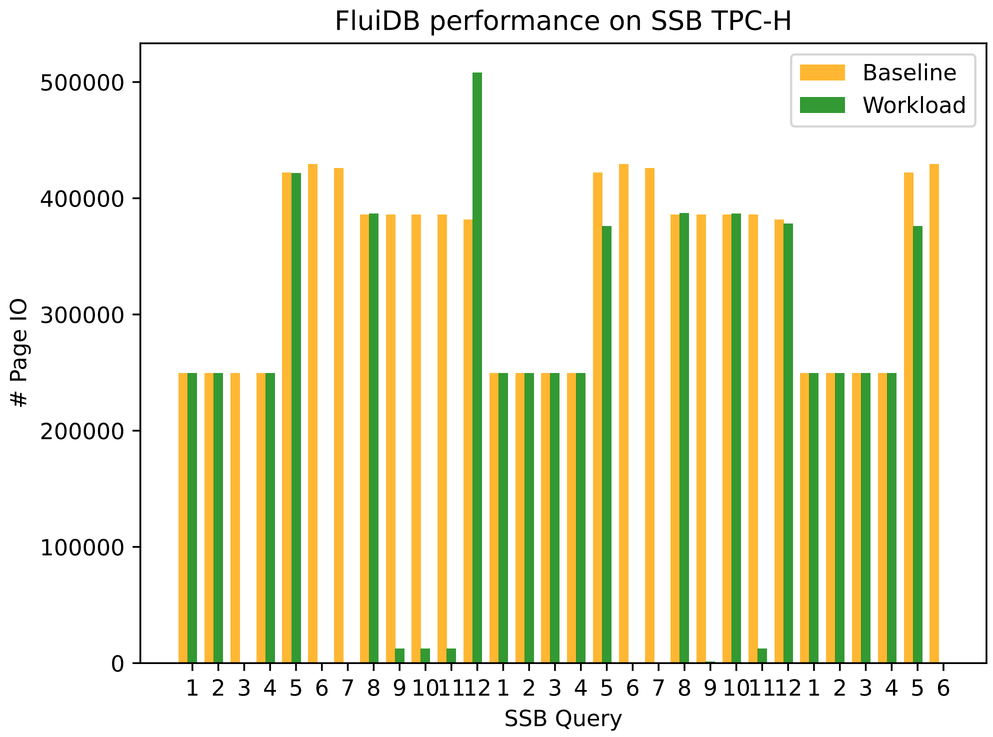


Figure 7.3: The total page read/writes for each query a sequence of queries. The SSB query number is $mod($query sequence$, 12$. The baseline is the query being run by FluiDB directly without any materialized nodes. The workload bars represent the cost of each query in a workload built within the same QDAG. The budget for this is the minimum budget within which FluiDB can run each individual query.

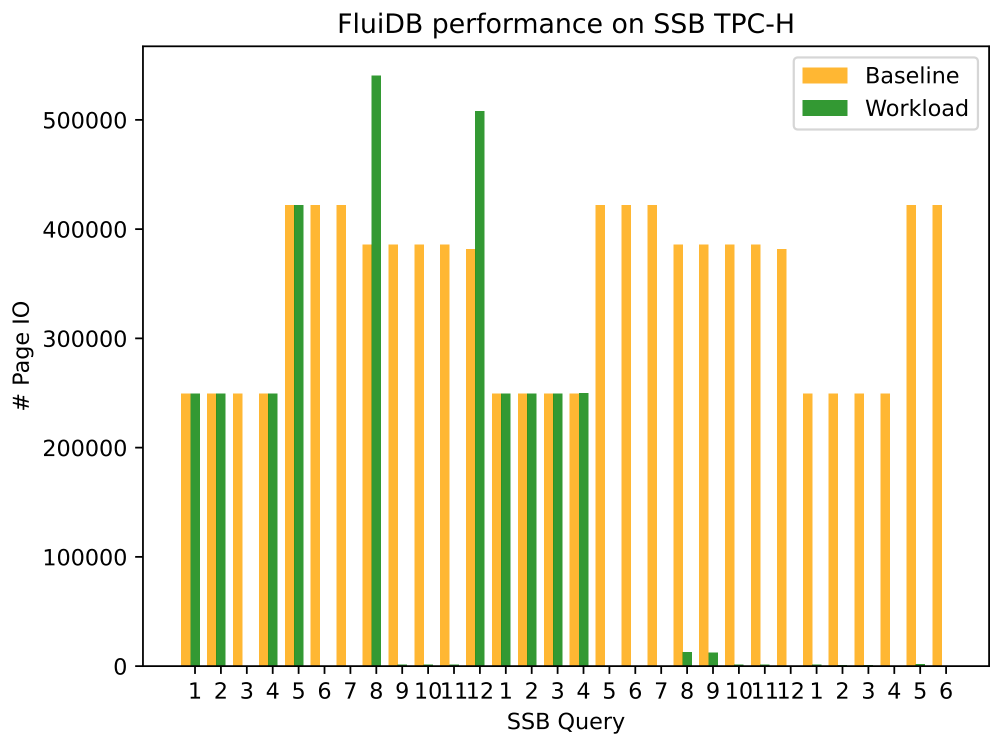


Figure 7.4: The total page read/writes for each query a sequence of queries. The SSB query number is $mod(\text{query sequence}, 12)$. The baseline is the query being run by FluiDB directly without any materialized nodes. The workload bars represent the cost of each query in a workload built within the same QDAG. The budget for this is triple the minimum budget within which FluiDB can run each individual query.

during query 5. In fact the next garbage collection is triggered during query 7, at a moment when $lineorder)$ is deletable, unprotected, and a prime candidate for deletion based on the GC heuristics.

Alas, when query 8 requires $lineorder)$ for its plan, FluiDB needs to re-construct it in the case of lax budgetary constraints but, not in the case of strict constraints.

This example of FluiDB being force to locally produce more expensive plans is an of FluiDB being more opportunistic the lower the available budget is, and more adventurous when operating with high budgets. When FluiDB is frugal it is generally prone to miss oportunities to share computation between queries. There are times however that this fugality saves it from bad heuristic-based decisions that it is allowed to make otherwise.

FluiDB aspires to deal with the entire workload as if it were planning a single query. While any decision during the planning of a single query can be undone, FluiDB is tragically forced to commit to whatever adveturous or conservative decisions it makes at the end of every planning iteration, doomed to pay dearly for every misstep but to reap the rewards of every insightful choice.

7.1 A note on size estimation

The size of the budget may feen fairly excessive for the size required for the primary tables. To understand why that is we need to look at the largest nodes in the QDAG:

Pages	Expr
188000	$lineorder$
547100	$customer \bowtie (date \bowtie lineorder)$
376200	$\sigma((customer \bowtie (date \bowtie lineorder)) \bowtie supplier)$
376200	$\sigma((date \bowtie lineorder)) \bowtie supplier)$
273600	$\sigma(customer \bowtie (date \bowtie lineorder))$
188100	$\sigma((\sigma(customer \bowtie (date \bowtie lineorder))) \bowtie supplier)$

From looking at those nodes it seems that a sufficiently advanced garbage collector should be able to support plans that materialize the nodes in about half the budget as fluidb should have been able to plan the query needing around double the pages of the largest equijoin which for us is customer date lineorder.

The main issue however, as it is with many query prolcessing systems [49], is the cardinality estimator. With a more sophisticated cardinality estimator the required pages.

The main issue with the cardinality estimator is that it assumes that a natural join there are no foreign key lookup failures, that is, that all natural joins are extension joins. Therefore

$$|\sigma_{p(customer)}(customer \bowtie (date \bowtie lineorder))| = |(\sigma_{p(customer)}customer \bowtie (date \bowtie lineorder))|$$

CHAPTER 8

Conclusions and future perspectives

We had our fun

(Someone Else)

Chapter summary

- Summary of FluiDB concepts.
- Future work.

In this thesis we coverd the important and interesting aspects of FluiDB. Implementing the system itself involved much work that is not covered in this thesis in order to not flood the reader with details that might remove focus from the important stuff. Some of these are:

- Implementation of a backtracking-based, caching pretty printing library.
- A query decorellation algorithm.
- A set of tools for debugging and tracing backtracking algoirhtms.

Aside from those, in the introduction we discussed some examples where data layout can be radically adapted to the workload by making use of reversible operations and integrating a garbage collector into the planner. We further introduced the relational database model and provided an overview of the various stages of query processing. We focused two subfields of database research that are of particular relevance to FluiDB: code generation for in-memory databases, and intermediate result recycling.

The first non-introductory chapter (chapter 3) goes in depth about the semantics of reversible operations and how they, along with possible intermediate results for each query form the QDAG. It covers topics that relate to the construction of the QDAG, the use of the QDAG for enumerating plans, higher levels of organization of the QDAG nodes into clusters and how those cluseters are used by FluiDB to infer properties of intermediate query results like cardinality and relation extent using propagators.

Chpater 4 on physical planning starts off by introducing a novel framework for implementing search algorithms, the `HCntT` monad transformer. Computationn expressed in terms of `HCntT` can dynamically switch between branches of the backtracking algorithm essentially implementing a weighted search. The main novelty lies in the implementatiuon of `once` and `fallback` operations that, to our knowledge, are not supported by any similar weighted search framework.

Based on `HCntT` we describe the implementation of the physical planner that lies at the core of FluiDB. The physical planner uses an $A^{\star}$-like algorithm to traverse the QDAG in order to enumerate query plans that take. Query plan fragments are interleaved with GC runs that produce sequences of table deletions when the planner detects that the storage budget is exceeded.

The decisions made by the planner are besd on computations that depend on the inventory of materialized relationions, namely variations of intermediate result cost estimation and intermediate result materializability. It makes sense to implement computations in terms of an intcremental computation framework as there are major opportunities for sharing between evaluations. Furthermore, the performance of these computations is sensitive to the order of evaluation due to absorbing elements and $\min$ operations. To solve these problems we developed *Antisthenis* which was described in detail in chapter 5.

We wrap up the technical description of FluiDB with chapter 6 that goes in depth about how FluiDB transpiles physical plans into C++ and what algorithms are used to implement each operation and their reverse.

Finally, we present some exerimental results to demonstrate the performance characteristics of FluiDB and any system that might implement similar concepts to it. Particularly, we run the SSB TPC-H benchmark and show that FluiDB can successfully outperform a similar system that only does per-query optimization in amortized cost and even in most per-query cases. We also demonstrate the expected result that FluiDB's repertoire of heuristics may cause individual queries planned within the context of a workload to be be less performant than the same query planned in isolation. We furthed demonstrate how FluiDB usually succeeds at making better use of larger storage budgets but occasionally, like every heuristic based system, will make decisions that hurt performance.

8.1 Future perspectives

FluiDB steps into a new path towards *truly* adaptive storage, but it does only a few steps. There are many interesting tangents to follow and may places where FluiDB does not take full advantage of the state of the art in database research.

Starting with the latter, the most important place where FluiDB can do better is cardinality estimation. As mention in section (7.1) FluiDB is very naive w.r.t. to how it estimates sizes. That said the propagator network structure used to estimate cardinalities can be augmented to propagate information about selectivity and statistics about each node. These statistice could be collected for every materialized intermediate relation to enrich the much needed information available during planning.

Another low hanging fruit is parallelism. All plans that FluiDB produces are single threaded but there is no fundamental reason for that to be so. A simple data dependency analysis could reveal opportunities for parallelism and operators themselves could be easily replaced with parallel versions. A more difficoult but very interesting problem would be how to teach the planner to take this into account in order to produce plans that aren't just accidentally parallel, but are designed taking such speedups into account.

Another important shortcoming of FluiDB as it was implemented is its complete lack of support for updates. There are heaps of literature to draw from and many excellent approaches to the problem. It would be interesting to find out which of these and how they could be applied to the peculiar case of FluiDB's data model where primary tables are likely not materialized in memory.

Furthermore, a system based on FluiDB could take advantage of non-relational operations like table compression, materialization of indexes, movement of data in a memory hierarchy in order to produce plans that can better take advantage of the various possibilities provided by the state of the art in database research.

Finally, a major bottleneck especially for cheap queries, is the latency of the C++ compiler that can take up to 10 sec to build a query plan into an executable. The FluiDB code generation make absolutely no attempt to make life easier for the C++ compiler, making heavy use of metaprogramming facilities like templates, traits, and **`constexpr`** expressions. There has been a lot of work to mitigate this problem, in the literature, some good examples of which are outlined in the background section 2.3.1

Bibliography

[1] Yehoshua Sagiv and Mihalis Yannakakis. “Equivalences Among Relational Expressions with the Union and Difference Operators”. In: *Journal of the ACM* 27.4 (Oct. 1980), pages 633–655.

[2] Donald D. Chamberlin et al. “A History and Evaluation of System R”. In: *Communications of the ACM* 24.10 (Oct. 1, 1981), pages 632–646.

[3] Christopher T. Haynes. “Logic Continuations”. In: *The Journal of Logic Programming* 4.2 (June 1, 1987), pages 157–176.

[4] G. Graefe and K. Ward. “Dynamic Query Evaluation Plans”. In: *ACM SIGMOD Record* 18.2 (June 1, 1989), pages 358–366.

[5] W. Pugh and T. Teitelbaum. “Incremental Computation via Function Caching”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’89*. The 16th ACM SIGPLAN-SIGACT Symposium. Austin, Texas, United States: ACM Press, 1989, pages 315–328.

[6] G. Graefe and W.J. McKenna. “The Volcano Optimizer Generator: Extensibility and Efficient Search”. In: *Proceedings of IEEE 9th International Conference on Data Engineering*. Proceedings of IEEE 9th International Conference on Data Engineering. Apr. 1993, pages 209–218.

[7] Goetz Graefe. “The Cascades Framework for Query Optimization”. In: *IEEE Data Eng. Bull.* 18.3 (1995), pages 19–29.

[8] Sheng Liang, Paul Hudak, and Mark Jones. “Monad Transformers and Modular Interpreters”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. New York, NY, USA: Association for Computing Machinery, Jan. 25, 1995, pages 333–343.

[9] Himanshu Gupta. “Selection of Views to Materialize in a Data Warehouse”. In: *Database Theory — ICDT ’97*. Edited by Foto Afrati and Phokion Kolaitis. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pages 98–112.

[10] Gérard Huet. “The Zipper”. In: *Journal of Functional Programming* 7.5 (Sept. 1997), pages 549–554.

[11] Dimitrios Theodoratos and Timos Sellis. “Data warehouse configuration”. In: *Proceedings of the 23rd International Conference on Very Large Databases, VLDB 1997*. 23rd International Conference on Very Large Databases, VLDB 1997. Morgan Kaufmann, Jan. 1, 1997, pages 126–135.

[12] Jeffrey D. Ullman. “Information Integration Using Logical Views”. In: *International Conference on Database Theory*. Springer, 1997, pages 19–40.

[13] Ralf Hinze. “Deriving Backtracking Monad Transformers”. In: *ACM SIGPLAN Notices* 35 (Aug. 15, 2000).

[14] Prasan Roy et al. “Efficient and Extensible Algorithms for Multi Query Optimization”. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 2000, pages 249–260.

[15] Hoshi Mistry et al. “Materialized View Selection and Maintenance Using Multi-Query Optimization”. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’01. New York, NY, USA: Association for Computing Machinery, May 1, 2001, pages 307–318.

[16] Michael Stillger et al. “LEO-DB2’s Learning Optimizer”. In: *VLDB*. Volume 1. 2001, pages 19–28.

[17] John Hughes. “Programming with Arrows”. In: *Advanced Functional Programming*. Edited by Varmo Vene and Tarmo Uustalu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pages 73–129.

[18] Tarmo Uustalu and Varmo Vene. *Comonadic Functional Attribute Evaluation*. Volume 6. Jan. 1, 2005, page 162. 145 pages.

[19] Gang Gou, J. X. Yu, and Hongjun Lu. “A/Sup */ Search: An Efficient and Flexible Approach to Materialized View Selection”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 3.36 (2006), pages 411–425.

[20] Jeremy E Dawson. “Compound Monads and the Kleisli Category”. In: *Unpublished note. Available online at http://users. cecs. anu. edu. au/∼ jeremy/pubs/cmkc* (2007).

[21] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. “A Pay-as-You-Go Framework for Query Execution Feedback”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pages 1141–1152.

[22] Conor McBride and Ross Paterson. “Applicative Programming with Effects”. In: *Journal of functional programming* 18.1 (2008), pages 1–13.

[23] Janis Voigtländer. “Asymptotic Improvement of Computations over Free Monads”. In: *Mathematics of Program Construction*. Edited by Philippe Audebaud and Christine Paulin-Mohring. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pages 388–403.

[24] Alexey Andreyevich Radul. “Propagation Networks: A Flexible and Expressive Substrate for Computation”. In: (2009).

[25] Gerald Jay Sussman and Alexey Radul. "The Art of the Propagator". In: (Jan. 26, 2009).

[26] Brent Yorgey. "The Typeclassopedia". In: (2009).

[27] Pradeep Kumar Gunda et al. "Nectar: Automatic Management of Data and Computation in Datacenters." In: *OSDI*. Volume 10. 2010, pages 1–8.

[28] Herodotos Herodotou and Shivnath Babu. "Xplus: A Sql-Tuning-Aware Query Optimizer". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pages 1149–1160.

[29] Milena G. Ivanova et al. "An Architecture for Recycling Intermediates in a Column-Store". In: *ACM Transactions on Database Systems (TODS)* 35.4 (2010), pages 1–43.

[30] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. "Generating Code for Holistic Query Evaluation". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). Mar. 2010, pages 613–624.

[31] Tomasz Nykiel et al. "MRShare: Sharing across Multiple Queries in MapReduce". In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 1, 2010), pages 494–505.

[32] Pramod Bhatotia et al. "Incoop: MapReduce for Incremental Computations". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. New York, NY, USA: Association for Computing Machinery, Oct. 26, 2011, pages 1–14.

[33] Swati V. Chande and Madhavi Sinha. "Genetic Optimization for the Join Ordering Problem of Database Queries". In: *2011 Annual IEEE India Conference*. IEEE, 2011, pages 1–5.

[34] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots". In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pages 195–206.

[35] Iman Elghandour and Ashraf Aboulnaga. *ReStore: Reusing Results of MapReduce Jobs*. Comment: VLDB2012. Feb. 29, 2012. arXiv: `1203.0061 [cs]`. URL: `http://arxiv.org/abs/1203.0061` (visited on 11/26/2021).

[36] Imene Mami and Zohra Bellahsene. "A Survey of View Selection Methods". In: *Acm Sigmod Record* 41.1 (2012), pages 20–29.

[37] Srinath Shankar et al. "Query Optimization in Microsoft SQL Server PDW". In: *Proceedings of the 2012 International Conference on Management of Data - SIGMOD '12*. The 2012 International Conference. Scottsdale, Arizona, USA: ACM Press, 2012, page 767.

[38] Lindsey Kuper and Ryan R. Newton. "LVars: Lattice-Based Data Structures for Deterministic Parallelism". In: *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC '13. New York, NY, USA: Association for Computing Machinery, Sept. 23, 2013, pages 71–84.

[39] Simon Marlow et al. “The Haxl Project at Facebook”. In: *Proceedings of the Code Mesh London* (2013).

[40] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. “Recycling in Pipelined Query Evaluation”. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pages 338–349.

[41] Guoping Wang and Chee-Yong Chan. “Multi-Query Optimization in Mapreduce Framework”. In: *Proceedings of the VLDB Endowment* 7.3 (2013), pages 145–156.

[42] Matthew A. Hammer et al. “Adapton: Composable, Demand-Driven Incremental Computation”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14: ACM SIGPLAN Conference on Programming Language Design and Implementation. Edinburgh United Kingdom: ACM, June 9, 2014, pages 156–166.

[43] Luis L. Perez and Christopher M. Jermaine. “History-Aware Query Optimization with Materialized Intermediate Views”. In: *2014 IEEE 30th International Conference on Data Engineering*. 2014 IEEE 30th International Conference on Data Engineering. Mar. 2014, pages 520–531.

[44] Luis L. Perez and Christopher M. Jermaine. “History-Aware Query Optimization with Materialized Intermediate Views”. In: *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pages 520–531.

[45] Mohamed A. Soliman et al. “Orca: A Modular Query Optimizer Architecture for Big Data”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 2014, pages 337–348.

[46] Mohamed A. Soliman et al. “Orca: A Modular Query Optimizer Architecture for Big Data”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14. New York, NY, USA: Association for Computing Machinery, June 18, 2014, pages 337–348.

[47] Michael Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. New York, NY, USA: Association for Computing Machinery, May 27, 2015, pages 1383–1394.

[48] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. “An Overview of Decision Support Benchmarks: TPC-DS, TPC-H and SSB”. In: *New Contributions in Information Systems and Technologies*. Edited by Alvaro Rocha et al. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2015, pages 619–628.

[49] Viktor Leis et al. “How Good Are Query Optimizers, Really?” In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015), pages 204–215.

[50] Jack Chen et al. "The MemSQL Query Optimizer: A Modern Optimizer for Real-Time Analytics in a Distributed Database". In: *Proceedings of the VLDB Endowment* 9.13 (Sept. 1, 2016), pages 1401–1412.

[51] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. "Sampling-Based Query Re-Optimization". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pages 1721–1736.

[52] Tarun Kathuria and S. Sudarshan. "Efficient and Provable Multi-Query Optimization". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2017, pages 53–67.

[53] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. "Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last". In: *Proceedings of the VLDB Endowment* 11.1 (Sept. 1, 2017), pages 1–13.

[54] William Arthur Rogers et al. "Multi-Query Optimization". Patent. US Patent App. 15/222,229. Mar. 23, 2017.

[55] E. Yu. Sharygin et al. "Query Compilation in PostgreSQL by Specialization of the DBMS Source Code". In: *Programming and Computer Software* 43.6 (Nov. 1, 2017), pages 353–365.

[56] Edmon Begoli et al. "Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pages 221–230.

[57] Bailu Ding et al. "Plan Stitch: Harnessing the Best of Many Plans". In: *Proceedings of the VLDB Endowment* 11.10 (June 2018), pages 1123–1136.

[58] Stavros Harizopoulos et al. "OLTP through the Looking Glass, and What We Found There". In: *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*. 2018, pages 409–439.

[59] Sven Keidel and Sebastian Erdweg. "Sound and Reusable Components for Abstract Interpretation". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), 176:1–176:28.

[60] Ryan Marcus et al. "Neo: A Learned Query Optimizer". In: *Proceedings of the VLDB Endowment* 12.11 (July 2019), pages 1705–1718.

[61] Jennifer Ortiz et al. *An Empirical Analysis of Deep Learning for Cardinality Estimation*. Sept. 11, 2019. arXiv: `1905.06425 [cs]`. URL: `http://arxiv.org/abs/1905.06425` (visited on 11/28/2021).

[62] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam Netherlands: ACM, June 25, 2019, pages 1981–1984.

[63] Ana Nunes Alonso et al. “Building a Polyglot Data Access Layer for a Low-Code Application Development Platform: (Experience Report)”. In: *Distributed Applications and Interoperable Systems*. Edited by Anne Remke and Valerio Schiavoni. Volume 12135. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pages 95–103.

[64] Daniel Selsam, Simon Hudon, and Leonardo deMoura. “Sealing Pointer-Based Optimizations behind Pure Functions”. In: *Proceedings of the ACM on Programming Languages* 4 (ICFP Aug. 2, 2020), 115:1–115:20.

[65] Chris Hanson and Gerald Jay Sussman. *Software Design for Flexibility: How to Avoid Programming Yourself into a Corner*. MIT Press, Mar. 9, 2021. 449 pages. Google Books: `iM_tDwAAQBAJ`.

[66] Donnacha Oisín Kidney and Nicolas Wu. “Algebras for Weighted Search”. In: *Proceedings of the ACM on Programming Languages* 5 (ICFP Aug. 18, 2021), 72:1–72:30.

[67] Guoliang Li, Xuanhe Zhou, and Lei Cao. “Machine Learning for Databases”. In: *The First International Conference on AI-ML-Systems*. AIMLSystems 2021. New York, NY, USA: Association for Computing Machinery, Oct. 21, 2021, pages 1–2.

[68] Pietro Michiardi, Damiano Carra, and Sara Migliorini. “Cache-Based Multi-Query Optimization for Data-Intensive Scalable Computing Frameworks”. In: *Information Systems Frontiers* 23.1 (2021), pages 35–51.

[69] Bartosz Milewski. *The Dao of Functional Programming*.

[70] 14:00-17:00. *ISO/IEC 14882:2020*. ISO. URL: `https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/93/79358.html` (visited on 10/28/2021).

[71] *Associated Type Synonyms | ACM SIGPLAN Notices*. URL: `https://dl.acm.org/doi/abs/10.1145/1090189.1086397?casa_token=HLcvyM59laEAAAAA:3yH9-bJAMmzqia0ewv_MReCDVnbfydL5nsIokMy4g8KWDC12huZV0_X7kMSVaE257aTivPkkuZLO0A` (visited on 10/10/2021).

[72] *Implementing Data Cubes Efficiently | Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. URL: `https://dlnext.acm.org/doi/10.1145/233269.233333` (visited on 11/26/2021).

[73] Oleg Kiselyov et al. “Backtracking, Interleaving, and Terminating Monad Transformers”. In: (), page 12.

[74] Edward Kmett. *Propagators*.

[75] Tim Kraska et al. “SageDB: A Learned Database System”. In: (), page 10.

[76] Daan Leijen. “Parsec: Direct Style Monadic Parser Combinators For The Real World”. In: (), page 22.

[77] Thomas Neumann. *Evolution of a CompilingQuery Engine*. URL: https://www.semanticscholar.org/paper/Evolution-of-a-CompilingQuery-Engine-Neumann/2cd16bb2d3680d04948f4536270c36d98fd4e768 (visited on 11/28/2021).

[78] Luis L. Perez and Christopher M. Jermaine. *History-Aware Query Optimization with Materialized Intermediate Views*.

[79] Chris Perivolaropoulos. *Fakedrake/Ssb-Dbgen*.

[80] Chris Perivolaropoulos. *Fakedrake/Ssb-Dbgen*.

[81] *Precompiled Headers (PCH)*. URL: https://releases.llvm.org/3.1/tools/clang/docs/PCHInternals.html (visited on 12/03/2021).