Deductive Optimization of Relational Data Storage

JOHN FESER, MIT, USA
SAM MADDEN, MIT, USA
NAN TANG, QCRI HBKU, Qatar
ARMANDO SOLAR-LEZAMA, MIT, USA

Optimizing the physical data storage and retrieval of data are two key database management problems. In this paper, we propose a language that can express both a relational query and the layout of its data. Our language can express a wide range of physical database layouts, going well beyond the row- and column-based methods that are widely used in database management systems. We use deductive program synthesis to turn a high-level relational representation of a database query into a highly optimized low-level implementation which operates on a specialized layout of the dataset. We build an optimizing compiler for this language and conduct experiments using a popular database benchmark, which shows that the performance of our specialized queries is better than a state-of-the-art in memory compiled database system while achieving an order-of-magnitude reduction in memory use.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Automatic programming*; • **Information systems** → *Relational database query languages*; **Data layout**.

Additional Key Words and Phrases: databases, deductive synthesis, data representation synthesis

ACM Reference Format:
John Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. 2020. Deductive Optimization of Relational Data Storage. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 170 (November 2020), 30 pages. https://doi.org/10.1145/3428238

1 INTRODUCTION

Traditional database systems are generic and powerful, but they are not well optimized for static databases. A static database is one where the data changes slowly or not at all and the queries are fixed. These two constraints introduce opportunities for aggressive optimization and specialization. This paper introduces Castor: a domain specific language and compiler for building static databases. Castor achieves high performance by combining query compilation techniques from state-of-the-art in-memory databases [Neumann 2011] with a new deductive synthesis approach for generating specialized data structures.

To better understand the scenarios that Castor supports, consider these two use cases. First, consider a company which maintains a web dashboard for displaying internal analytics from data that is aggregated nightly. The queries used to construct the dashboard cannot be precomputed directly, because they depend on parameters like dates or customer IDs, but all the queries are generated from a few query templates. Additionally, not all of the data in the original database is needed, and some attributes are only used in aggregates. As another example, consider a company which is shipping

Authors' addresses: John Feser, CSAIL, MIT, 32 Vassar St., Cambridge, MA, 02139, USA, feser@csail.mit.edu; Sam Madden, CSAIL, MIT, 32 Vassar St., Cambridge, MA, 02139, USA, madden@csail.mit.edu; Nan Tang, Qatar Computing Research Institute, QCRI HBKU, Doha, Qatar, ntang@hbku.edu.qa; Armando Solar-Lezama, CSAIL, MIT, 32 Vassar St., Cambridge, MA, 02139, USA, asolar@csail.mit.edu.

2475-1421/2020/11-ART170
https://doi.org/10.1145/3428238

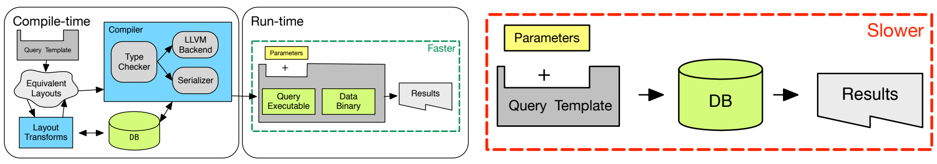


Fig. 1. An overview of the Castor system (left) vs a traditional RDBMS (right).

a GPS device that contains an embedded map. The map data is infrequently updated, and the device queries it in only a few specific ways. The GPS manufacturer cares more about compactness and efficiency than about generality. As with the company building the dashboard, it is desirable to produce a system that is optimal for the particular dataset to be stored.

These two companies could use a traditional database system, but using a system designed to support arbitrary queries will miss important optimization opportunities. Alternatively, they could implement their queries using custom data structures. This will give them tight control over their data layout and query implementation but will be difficult to develop and expensive to maintain.

Castor is an attempt to capture some of the optimization opportunities of static databases and to address the needs of these two scenarios. As Figure 1 illustrates, the input to Castor is a dataset and a parameterized query that a client will want to invoke on the data. The user then uses Castor's automatic optimizer or manually applies its high-level query transformations to generate an efficient implementation of an in-memory datastore specialized for the dataset and the parameterized query. The transformations available in Castor give the programmer tight control over the organization of the data in memory, allowing the user to trade off memory usage against query performance without the risk of introducing bugs. Castor uses code generation techniques from high-performance in-memory databases to produce the low-level implementations required for efficient execution. The result is a package of data and code that uses significantly less memory than the most efficient in-memory databases and for many queries can surpass the performance of in-memory databases that already rely on aggressive code generation and optimization [Neumann 2011].

1.1 Contributions

Our primary contribution is the layout algebra: a new notation to jointly represent the layout of data in memory and the queries that will be computed on it. This joint representation allows us to write transformations that manipulate both the layout and the query in a single rewrite rule. This makes it easy to apply aggressive layout transformations.

We describe a set of deductive optimization rules for the layout algebra that generalize traditional query optimization rules to jointly optimize the query and the data layout and an automatic optimizer that applies these rules. We also implement a specializing layout compiler that produces both a binary representation of the data from the high-level data representation and machine code for accessing it.

Integrated Layout & Query Language. We define the *layout algebra*, which extends the relational algebra [Codd 1970] with layout operators that describe the particular data items to be stored and the layout of that data in memory. The layout algebra is flexible and can express many layouts, including row stores and clustered indexes. It supports nesting layouts, which gives control over data locality and supports prejoining of data. Our use of a language which combines query and layout operators makes it possible to write deductive transformations that change both the runtime query behavior and the data layout.

Automated Deductive Optimizer. Castor provides a set of equivalence preserving transformations which can change both the query and the data layout. The user can use Castor's optimizer to automatically select a sequence of transformations to deductively optimize their query. Alternatively, they can apply transformations manually to optimize without worrying about introducing bugs. Castor's notation turns transformations that would be complex and global in other database systems into local syntactic changes.

Type-driven Layout Compiler. Existing relational synthesis tools use standard library data structures and make extensive use of pointer based data structures that hurt locality [Hawkins et al. 2011; Loncaric et al. 2018, 2016]. Castor uses a specializing layout compiler that takes the properties of the data into account when serializing it. Before generating the layout, Castor generates an abstraction called a *layout shape* which guides the layout specialization. For example, if the layout is a row-store with fixed-size tuples, the layout compiler will not emit a length field for the tuples. Instead, this length will be compiled directly into the query. This specialization process creates very compact datasets and avoids expensive branches in generated code.

High Performance Query Compiler. Castor uses code generation techniques from the high performance in-memory database literature [Neumann 2011; Rompf and Amin 2015; Shaikhha et al. 2016; Tahboub et al. 2018]. It eschews the traditional iterator based query execution model [Graefe 1994] in favor of a code generation technique that produces simple, easily optimized low-level code. Castor directly generates LLVM IR and augments the generated IR with information about the layout that allows LLVM to further optimize it.

Empirical Evaluation. We empirically evaluate Castor on a benchmark derived from TPC-H, a standard database benchmark [Council 2008]. We show that Castor is competitive with the state of the art in-memory compiled database system Hyper [Neumann 2011] while using significantly less memory. We also show that Castor scales to larger queries than the leading data-structure synthesis tool Cozy [Loncaric et al. 2018].

1.2 Limitations

Castor constructs read-only databases. This design decision limits the appropriate use cases for Castor but it enables important optimizations. Castor takes advantage of the absence of updates to tightly pack data together, which improves locality. Castor also aggressively specializes the compiled query by including information about the layout, such as lengths of arrays and offsets of layout structures. Providing this information to the compiler improves the generated code.

The optimizer processes one query at a time. Castor supports multiple-query workloads by reducing them to single-query workloads. However, the optimizer does not contain transformations that exploit possible sharing of layouts between different parts of a query, so the optimizer may replicate more data than necessary. However, Castor removes any data which is not needed by the query and it produces compact layouts for the data that remains, which reduces the overhead of any replication.

2 MOTIVATING EXAMPLE

We now describe the operation of Castor on an application from the software engineering literature. DemoMatch is a tool which helps users understand complex APIs using software demonstrations [Yessenov et al. 2017]. DemoMatch maintains a database of program traces—computed offline—which it queries to discover how to use an API. DemoMatch is a good fit for Castor because: (1) computing new traces is an infrequent task so the data in question is largely static and (2) the data is automatically queried by the tool, so there is no need to support ad-hoc queries. Finally, query performance is important for DemoMatch to work interactively.

2.1 Background

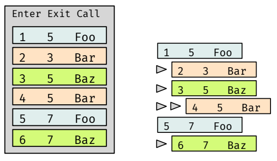
DemoMatch stores program traces as ordered collections of *events* (e.g., function calls). Traces have an inherent tree structure: each event has an `enter` and an `exit` and nested events may occur between the enter and exit. Figure 2 shows the table and tree structure of the DemoMatch data.

A critical query in the DemoMatch system finds nested calls to particular functions in a trace of program events:

select $p.enter$, $c.enter$ from log as p, log as c where

$p.enter < c.enter \wedge c.enter < p.exit \wedge p.id = \$pid \wedge c.id = \$cid$

We refer to the caller as the *parent* function and the callee as the *child* function. Let p and c be the traces of events inside the parent and child function bodies respectively. The join predicate $p.enter < c.enter \wedge c.enter < p.exit$ selects calls to the child function from inside the parent function. The predicate $p.id = \$pid \wedge c.id = \$cid$ selects the pair of functions that we are interested in, where $\$pid$ and $\$cid$ are parameters.



(a) In tabular form. (b) In tree form.

Fig. 2. Graphical representation of the DemoMatch data.

2.2 The Layout Algebra

Castor programs are written in a language called the layout algebra. The layout algebra is similar to the relational algebra, but as we will see shortly, it can represent the layout of data as well as the operation of queries. By design, it is more procedural than SQL, which is more akin to the relational calculus [Codd 1971]. For example, SQL leaves choices like join ordering to the query planner, whereas in the layout algebra join ordering is explicit.

In designing the layout algebra, we follow a well-worn path in deductive synthesis of creating a uniform representation that can capture all the refinement steps from a high-level program to a low-level one. Accordingly, the layout algebra can express programs which contain a mixture of high-level relational constructs and low-level layout constructs. At some point, a layout algebra program contains enough implementation information that the compiler can process it. We say that these programs are *well-staged* (Section 3.4).

Here is the nested call query from Section 2.1 translated into the layout algebra:

$$\begin{array}{l}
\texttt{select}(\{enter_p, enter_c\}, \texttt{join}(enter_p < enter_c \wedge enter_c < exit_p, \\
\quad \texttt{filter}(\$pid = id_p, \texttt{select}(\{id \mapsto id_p, enter \mapsto enter_p, exit \mapsto exit_p\}, log)), \\
\quad \texttt{filter}(\$cid = id_c, \texttt{select}(\{id \mapsto id_c, enter \mapsto enter_c\}, log))))
\end{array}$$

There are three layout algebra operators in this query. `filter`(p, r) filters the relation r by the predicate p. `join`(p, r, r') takes the cross product of relations r and r' and filters the result by p. `select` takes a list of expressions with optional names and a query, and selects the value of each expression for each tuple in the query, possibly renaming it.

The scoping rules for the layout algebra may look somewhat unusual, but they are intended to mimic the scoping conventions of SQL. In this query, the names *enter*, *exit* and *id* are field names in the *log* relation. The `select` operators introduce new names for these fields, using the $\mapsto$ operator, so that *log* can be joined with itself. We formalize the semantics of the layout algebra, including the scoping rules, in Section 3.2.

Note that at this point no layout is specified for *log*, so this program is not well-staged, and so cannot be compiled. However, it still has well defined semantics. In Sections 2.4 to 2.6, we describe how layouts can be incrementally introduced by transforming the program until it is well-staged.

2.3 Optimization Trade-Offs

The nested call query is interesting because the data in question is fairly large—hundreds of thousands of rows—and keeping it fully in memory, or even better in cache, is a significant performance win. Therefore, minimizing the size of the data in memory should improve performance.

However, there is a fundamental trade-off between a more compact data representation and allowing for efficient access. Sometimes the two goals are aligned, but often they are not. For example, creating a hash index allows efficient access using a key, but introduces overhead in the form of a mapping between hash keys and values.

In the rest of this section we examine three layouts at different points in this trade-off space: a compact nested layout with no index structures (Figure 3a), a layout based on a single hash index (Figure 3b), and a layout based on a hash index and an ordered index (Figure 3c). A priori, none of these layouts is clearly superior. The hash based layout is the largest, but has the best lookup properties. The nested layout precomputes the join and uses nesting to reduce the result size, but is more expensive for lookups. The last layout must compute the join at runtime but it has indexes that will make that computation fast. The power of Castor is that it allows users to effectively explore different layout trade-offs by freeing them from the need to ensure the correctness of each candidate.

2.4 Nested Layout

Our first approach to optimizing the nested call query is to materialize the join, since joins are usually expensive, and to use nesting to reduce the size of the resulting layout.

The first step is to apply transformation rules (Section 4.2) to hoist and merge the filters. Now the join is in a term with no query parameters, so it can be evaluated at compile time:

$$\begin{array}{l}
\texttt{select}(\{enter_p, enter_c\}, \texttt{filter}(\$pid = id_p \wedge \$cid = id_c, \\
\quad \texttt{join}(enter_p < enter_c \wedge enter_c < exit_p, \\
\quad\quad \texttt{select}(\{id \mapsto id_p, enter \mapsto enter_p, exit \mapsto exit_p\}, log), \\
\quad\quad \texttt{select}(\{id \mapsto id_c, enter \mapsto enter_c\}, log))))
\end{array}$$

After applying two more rules—projection to eliminate unnecessary fields (Section 4.3) and join elimination (Section 4.6)—the result is the following *layout program* (represented graphically in Figure 3a):

$$\begin{array}{l}
\texttt{select}(\{enter_p, enter_c\}, \texttt{filter}(id_c = \$cid \wedge id_p = \$pid, \\
\quad \mathbf{list}(\texttt{select}(\{id \mapsto id_p, enter \mapsto enter_p, exit \mapsto exit_p\}, log)\ \texttt{as}\ lp, \\
\quad\quad\quad \mathbf{tuple}_{\text{cross}}([\mathbf{scalar}(lp.id_p), \mathbf{scalar}(lp.enter_p), \\
\quad\quad\quad\quad\quad\quad \mathbf{list}(\texttt{filter}(lp.enter_p < enter_c \wedge enter_c < lp.exit_p, \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{select}(\{id \mapsto id_c, enter \mapsto enter_c\}, log))\ \texttt{as}\ lc, \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{tuple}_{\text{cross}}([\mathbf{scalar}(lc.id_c), \mathbf{scalar}(lc.enter_c)])) \\
\quad\quad\quad\quad\quad\quad]) \\
\quad\quad\quad)))
\end{array}$$

In this program we see our first layout operators: $\mathbf{list}(\cdot,\cdot)$ and $\mathbf{tuple}_{\text{cross}}([\ldots])$.[1] The layout algebra extends the relational algebra with these operators, allowing us to write layout expressions, which describe how their arguments will be laid out in memory.

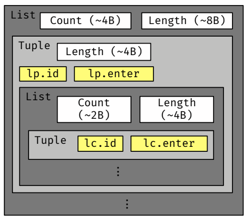
The above program can be read as follows. The operator $\mathbf{list}(q\ \texttt{as}\ l, q')$ creates a list with one element for every tuple in q and each element in the list is laid out according to q'. The outermost **list** in the program selects the *id*, *enter* and *exit* fields of the *log* relation and lays out each element

[1] As a point of notation, we separate layout operators from non-layout operators visually by **bolding** them. This is just to make the programs easier to read.

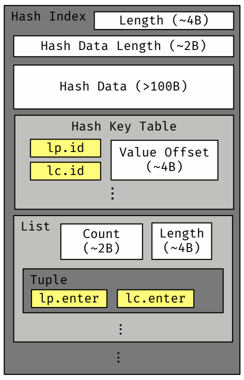
of the list as a $\texttt{tuple}_{\text{cross}}$[2]. The first two elements in the tuple are the scalar representations of the id_p and $enter_p$ fields, and the third element is a nested list. Note that the content of that inner list is filtered based on the value of $lp.enter_p$ and $lp.exit_p$, and each element is laid out as a pair of two scalars id_c and $enter_c$.

The query is now well-staged because it satisfies the rules in Section 3.4. At a high-level, the rules require that we never use a relation without specifying its layout, a requirement that is satisfied in this case because all references to the log relation appear in the first arguments of **list** operators.

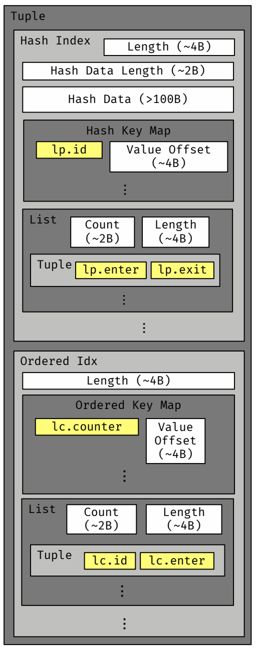
Figure 3a shows the structure of the resulting layout. This layout is quite compact. It is smaller than the fully materialized join because of the nesting; the caller `id` and `enter` fields are only stored once for each matching callee record. When we benchmark this query, we find that it performs reasonably well (11.5ms) and is fairly small (50Mb).



(a) Nested.



(b) Hash-index.



(c) Ordered-index.

Fig. 3. Layouts for the DemoMatch queries. The yellow boxes contain relational data, the white boxes contain metadata, and the gray boxes are the layout structure.

2.5 Hash-Index Layout

Now we optimize for lookup performance by fully materializing the join and creating a hash index. This layout will be larger than the nested layout but lookups into the hash index will be quick, which will make evaluating the equality predicates on id fast. Figure 3b shows the structure of the resulting layout. When we evaluate the query, we find that it is much faster (0.4ms) but is larger than the nested query (60Mb).

2.6 Hash- & Ordered-Index Layout

Finally, we investigate a layout (Figure 4) which avoids the full join materialization, but still has enough indexing to be fast. We can see that the join condition is a range predicate, so we would like to use an index that supports efficient range queries to make that predicate efficient (Section 4.5). Then we can push the filters and introduce a hash table to select id_p. The resulting layout is shown in Figure 3c. This layout will be larger than the original relation, but smaller than the other two layouts (9.8Mb), and it allows for much faster computation of the join and one of the filters (0.6ms).

This program introduces three new operators: **ordered-idx**, **hash-idx** and `depjoin`. **ordered-idx** creates indexes that support efficient range queries. It takes four parameters: *keys*, *values*, *upper*, and *lower*. *keys* is a relation that defines the set of keys and *values* is a dependent relation that defines the layout of values as a function of the keys. *lower* and *upper* are the bounds to use when reading the index ($p.enter_p$ and $p.exit$ in this case). **hash-idx**(*keys*, *values*, *lookup*) is similar,

[2]In this expression, cross specifies how the tuple will eventually be read. Layout operators evaluate to sequences, so a tuple needs to specify how these sequences should be combined. In this case, we take a cross product.

```
select({enter_p, enter_c},
  depjoin(hash-idx(select({id}, log) as h,
                   list(filter(h.id=id∧enter > exit, log) as lh,
                        tuple_cross([scalar(lh.enter↦enter_p), scalar(lh.exit)])),
                  $pid) as p,
          filter(id=$cid,
            ordered-idx(select({enter}, log) as o,
                        list(filter(enter=o.enter, log) as lo,
                             tuple_cross([scalar(lo.id), scalar(lo.enter↦enter_c)])),
                        p.enter_p, p.exit))))
```

Fig. 4. A layout that combines hash- and ordered-indexes.

but it creates efficient point indexes using hash tables. *lookup* is the key to look up in the index (in this query, the key is $\$pid$).

The more interesting operator is the dependent join operator `depjoin`. In a dependent join, the right-hand-side of the join can refer to fields from the left-hand-side. In the `depjoin` operator, the left-hand-side is given a name (here it is p) that the right-hand-side can use to refer to its fields. One way to think about a dependent join is as a relational `for` loop: it evaluates the right-hand-side for each tuple in the left-hand-side, concatenating the results. Unlike the layout operators **`list`**, **`hash-idx`** and **`ordered-idx`**, `depjoin` executes entirely at runtime. It does not introduce any layout structure.

3 LANGUAGE

In this section we describe the layout algebra in detail. The layout algebra starts with the relational algebra and extends it with layout operators. These layout operators have relational semantics, but they also have layout semantics which describes how to serialize them to data structures. The combination of relational and layout operators allows the layout algebra to express both a query and the data store that supports the execution of the query.

Programs in the layout algebra have three semantic interpretations:

(1) The *relational semantics* describes the behavior of a layout algebra program at a high level. We define this semantics using a theory of ordered finite relations [Cheung et al. 2013]. According to this semantics, a layout algebra program can be evaluated to a relation in a context containing relations and query parameters.
(2) The *layout semantics* describes how the compiler creates a data file from a well-staged layout algebra program. The layout semantics operates in a context which contains relations, but *not* query parameters.
(3) The *runtime semantics* describes how the compiled query executes, reading the layout file and using the query parameters to produce the query output. The runtime semantics operates in a context which contains query parameters but *not* relations.

These three semantics are connected: the layout semantics and the runtime semantics combine to implement the relational semantics. The relational semantics serves as a specification. An interpreter written according to the relational semantics should execute layout algebra programs in the same way as our compiler.

$$
\begin{aligned}
&x ::= \text{identifier} \quad o ::= \text{asc} \mid \text{desc} \quad \tau ::= \text{cross} \mid \text{concat}\\
&v ::= \text{integers} \mid \text{strings} \mid \text{Booleans} \mid \text{floats} \mid \text{dates} \mid \text{null}\\
&e ::= v \mid x \mid x.x \mid e+e' \mid e-e' \mid e\times e' \mid e/e' \mid e \,\%\, e' \mid e<e' \mid e\leq e' \mid e>e' \mid e\geq e' \mid e=e'\\
&\quad\mid \ \mathtt{if}\ e\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \mid \mathtt{exists}(q) \mid \mathtt{first}(q) \mid \mathtt{count}() \mid \mathtt{sum}(e) \mid \mathtt{min}(e) \mid \mathtt{max}(e) \mid \mathtt{avg}(e)\\
&t ::= \{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}\\
&q ::= \emptyset \mid x \mid \mathtt{dedup}(q) \mid \mathtt{select}(t, q) \mid \mathtt{filter}(e, q) \mid \mathtt{join}(e, q, q') \mid \mathtt{group\text{-}by}(t, [x_1, \ldots, x_m], q)\\
&\quad\mid \ \mathtt{order\text{-}by}([e_1\ o_1, \ldots, e_m\ o_m], q) \mid \mathtt{depjoin}(q\ \mathtt{as}\ x, q') \mid \mathbf{scalar}(x \mapsto e)\\
&\quad\mid \ \mathbf{tuple}_\tau(t) \mid \mathbf{list}(q_r\ \mathtt{as}\ x, q) \mid \mathbf{hash\text{-}idx}(q_k\ \mathtt{as}\ x, q_v, e_k) \mid \mathbf{ordered\text{-}idx}(q_k\ \mathtt{as}\ x, q_v, e_{lo}, e_{hi})
\end{aligned}
$$

Fig. 5. Syntax of the layout algebra.

3.1 Syntax

Figure 5 shows the syntax of the layout algebra. Note that the layout algebra can be divided into relational operators (`select`, `filter`, `join`, etc.) and layout operators (**`list`**, **`hash-idx`**, etc.). The layout algebra is a strict superset of the relational algebra. In fact, the layout operators have relational semantics in addition to byte-level data layout semantics (see Section 3.2.2).

3.2 Relational Semantics

The semantics (Figure 6) operates on three kinds of values: scalars, tuples and relations. Scalars are values like integers, Booleans, and strings. Tuples are finite mappings from field names to scalar values. Relations are represented as finite, ordered sequences of tuples. [ ] stands for the empty relation, : is the relation constructor, and ++ denotes the concatenation of relations.

We use sequences instead of sets for two reasons. First, sequences are more like bag semantics than the set semantics of the original relational algebra. This choice brings the layout algebra more in line with the semantics of SQL, which is convenient for our implementation. Second, sequences allow us to represent query outputs which have an ordering.

In the semantic rules, σ is an evaluation context; it maps names to scalar values. δ is a relational context; it maps names to relations. We separate the two contexts because the relational context δ is global and immutable; it consists of a universe of relations that exist when the query is executed (or compiled) which are contained in some other database system. The evaluation context σ initially contains the query parameters, but some operators introduce new bindings in σ. $\cup$ denotes the binding of a tuple into an evaluation context. Read $\sigma \cup t$ as a new evaluation context that contains the fields in t in addition to the names already in σ.

In the rules, $\vdash$ separates contexts and expressions and $\Downarrow$ separates expressions and results. Read $\sigma, \delta \vdash l \Downarrow s$ as "the layout l evaluates to the relation s in the context σ, δ."

We borrow the syntax of list comprehensions to describe the semantics of the layout algebra operators. For example, consider the list comprehension in the `filter` rule: $[t \mid t \leftarrow r_q \quad \sigma \cup t, \delta \vdash e \Downarrow \mathtt{true}]$, which corresponds to the expression $\mathtt{filter}(e, q)$. This list comprehension filters r_q by the predicate e where r_q is the relation produced by q. e is evaluated in a context $\sigma \cup t$ for each tuple t in r_q. Comprehensions that contain multiple $\leftarrow$, as in the `join` rule, should be read as the cross product that produces $[(x_1,y_1),(x_1,y_2),(x_2,y_1),(x_2,y_2)]$ from $[x_1,x_2]$ and $[y_1,y_2]$.

Finally, $\textsc{schema}(\cdot)$ is a function from a layout q to the set of field names in the output of q.

3.2.1 *Relational Operators.*

First, we describe the semantics of the relational operators: `filter`, `join`, `select`, `group-by`, `order-by`, `dedup`, and `depjoin`. These operators are modeled after their equivalent SQL constructs. For brevity and because they are straightforward, we omit the rules for `join`, `select`, `group-by`, `order-by`, and `dedup` from Figure 6.

$$Id = (Scope?, Name) \quad Context = Tuple = \{Id \mapsto Value\} \quad Relation = [Tuple]$$
$$\sigma : Context \quad \delta : Id \mapsto Relation \quad s : Id \quad t : Tuple \quad v : Value \quad r : Relation$$

$$\text{E-Tuple}\,\frac{t = \{n_1 \mapsto e_1, \ldots, n_m \mapsto e_m\} \qquad \forall i.\ \sigma, \delta \vdash e_i \Downarrow v_i}{\sigma, \delta \vdash t \Downarrow \{n_1 \mapsto v_1, \ldots, n_m \mapsto v_m\}} \qquad \text{R-Relation}\,\frac{(x, r) \in \delta}{\sigma, \delta \vdash x \Downarrow r} \qquad \text{R-Empty}\,\frac{}{\sigma, \delta \vdash \emptyset \Downarrow [\,]}$$

$$\text{R-Filter}\,\frac{\sigma, \delta \vdash q \Downarrow r_q \qquad r = [t \mid t \leftarrow r_q \quad \sigma \cup t, \delta \vdash e \Downarrow \mathtt{true}]}{\sigma, \delta \vdash \mathtt{filter}(e, q) \Downarrow r}$$

$$\text{R-Depjoin}\,\frac{\sigma, \delta \vdash q \Downarrow r \qquad r'' = \left[t' \;\middle|\; t \leftarrow r \quad t' \leftarrow r' \quad \sigma \cup t_s, \delta \vdash q' \Downarrow r' \quad t_s = \{s.f \mapsto v \mid (f \mapsto v) \in t\} \right]}{\sigma, \delta \vdash \mathtt{depjoin}(q \text{ as } s, q') \Downarrow r''}$$

$$\text{R-Scalar}\,\frac{\sigma, \delta \vdash e \Downarrow v}{\sigma, \delta \vdash \mathbf{scalar}(n \mapsto e) \Downarrow [\{n \mapsto v\}]} \qquad \text{R-List}\,\frac{\sigma, \delta \vdash \mathtt{depjoin}(q_r \text{ as } s, q) \Downarrow r}{\sigma, \delta \vdash \mathbf{list}(q_r \text{ as } s, q) \Downarrow r}$$

$$\text{R-OrderedIndex}\,\frac{\begin{array}{c}\sigma, \delta \vdash \mathtt{depjoin}(q_k \text{ as } s, \mathtt{filter}(v_{lo} \leq s.x \leq v_{hi}, q_v)) \Downarrow r \\ \sigma, \delta \vdash l_{lo} \Downarrow v_{lo} \qquad \sigma, \delta \vdash l_{hi} \Downarrow v_{hi} \qquad \textsc{schema}(q_k) = [x]\end{array}}{\sigma, \delta \vdash \mathbf{ordered\text{-}idx}(q_k \text{ as } s, q_v, l_{lo}, l_{hi}) \Downarrow r}$$

$$\text{R-Tuple1}\,\frac{}{\sigma, \delta \vdash \mathbf{tuple}_\tau([\,]) \Downarrow [\,]} \qquad \text{R-Tuple2}\,\frac{\sigma, \delta \vdash q_1 \Downarrow r_q \qquad \sigma, \delta \vdash \mathbf{tuple}_{\text{cross}}([q_2, \ldots, q_n]) \Downarrow r_{qs}}{\sigma, \delta \vdash \mathbf{tuple}_{\text{cross}}([q_1, \ldots, q_n]) \Downarrow [t \cup ts \mid t \leftarrow r_q \quad ts \leftarrow r_{qs}]}$$

Fig. 6. Selected relational semantics of the layout algebra. $\sigma, \delta \vdash l \Downarrow s$ means l evaluates to s in σ, δ.

`filter` filters a relation by a predicate e. `join` takes the cross product of two relations and filters it using a predicate e.

`select` is used for projection, aggregation, and renaming fields. It takes a tuple expression t and a relation r. If t contains no aggregation operators, then a new tuple will be constructed according to t for each tuple in r. If t contains an aggregation operator (`count`, `sum`, `min`, `max`, `avg`), then `select` will aggregate the rows in r. If t contains both aggregation and non-aggregation operators, then the non-aggregation operators will be evaluated on an arbitrary tuple in r.

`group-by` takes a list of expressions, a list of fields, and a relation. It groups the tuples in the relation by the values of the fields, then computes the aggregates in the expression list. `order-by` takes a list of expression-order pairs and a relation. It orders the tuples in the relation using the list of expressions to compute a key. `dedup` removes duplicate tuples.

Finally, `depjoin` denotes a dependent join, where the right-hand-side of the join can depend on values from the left-hand-side. It is similar to a for-each loop; $\mathtt{depjoin}(q \text{ as } n, q')$[3] can be read "evaluate q' for each tuple in q and concatenate the results." We use `depjoin` as a building block to define the semantics of the layout operators.

3.2.2 Layout Operators. The novelty of the layout algebra is that it can express the layout of data in addition to queries over that data. We introduce the following operators for describing data layouts: **`scalar`**, **`tuple`**, **`list`**, **`hash-idx`**, and **`ordered-idx`**. We chose these layout primitives because they are compositional, have good spatial locality and support common query patterns such as range and equality predicates.

The layout operators have relational semantics; we show the semantics of selected operators in Figure 6. Although the layout operators can be used to construct complex, nested layouts, they

[3] In this expression, n is a *scope*, and it qualifies the names in q. Scopes are discussed in more detail in Section 3.2.3.

evaluate to flat sequences of tuples of scalars, just like the relational operators. The rules in Figure 6 only describe the relational behavior of the layout operators; they do not address the question of how data is laid out or how it is accessed. We discuss these aspects of the layout operators in Section 3.3.

The simplest layout operators are **`scalar`** and **`tuple`**. Evaluating a **`scalar`** operator produces a relation containing a single tuple. The **`tuple`** operator represents a fixed-size, heterogeneous list of layouts. When evaluated, each layout in the tuple produces a relation, which are combined either with a cross product or by concatenation.

Note that evaluating a **`tuple`** operator produces a *relation* not a tuple. Although these semantics are slightly surprising, there are two reasons why we chose this behavior. First, it is consistent with the other layout operators, all of which evaluate to relations. Second, **`tuple`**s can contain other layouts (**`list`**s for example) which themselves evaluate to relations.

The remaining layout operators—**`list`**, **`hash-idx`** and **`ordered-idx`**—have a similar structure. We discuss the **`list`** operator in detail. **`list`** is essentially an alias for `depjoin`. Like `depjoin`, **`list`** takes two arguments: q_r and q. These two arguments should be interpreted as follows: q_r describes the data in the list. Each element of the list has a corresponding tuple in q_r, so the length of the list is the same as the length of q_r. One can think of each tuple in q_r as a kind of key that determines the contents of each list element. On the other hand, q describes how each list element is laid out. q will be evaluated separately for each tuple in q_r. It determines for each "key" in q_r, what the physical layout of each list element will be, as well as how that element must be read.

Returning to the query in Section 2.4, the inner **`list`** operator

$$\textbf{list}(\texttt{filter}(lp.enter_p < enter_c \wedge enter_c < lp.exit_p, \texttt{select}(\{id \mapsto id_c, enter \mapsto enter_c\}, log))\ \texttt{as}\ lc,$$
$$\textbf{tuple}_{\text{cross}}([\textbf{scalar}(lc.id_c), \textbf{scalar}(lc.enter_c)]))$$

selects the tuples in log where $enter_c$ is between $enter_p$ and $exit_p$, and creates a list of these tuples. The first argument describes the contents of the list and the second describes their layout. This program will generate a layout that has a list of tuples, structured as $[(id_{c1}, enter_{c1}),...,(id_{cn}, enter_{cn})]$.

`hash-idx` and **`ordered-idx`** are similar to **`list`**. They have a query q_k that describes which keys are in the index and a query q_v that describes the contents and layout of the values in the index.

For example, in:

$$\textbf{hash-idx}(\texttt{select}(\{id\}, log)\ \texttt{as}\ h,$$
$$\textbf{list}(\texttt{filter}(\{id = h.id\}, log)\ \texttt{as}\ l,$$
$$\textbf{tuple}_{\text{cross}}([\textbf{scalar}(l.enter), \textbf{scalar}(l.exit)])), \$pid)$$

the keys to the hash-index are the id fields from the log relation. For each of these fields, the index contains a list of corresponding $(enter, exit)$ pairs, stored in a tuple. When the hash-index is accessed, $\$pid$ is used as the key. This program generates a layout of the form: $\{id \mapsto [(enter, exit),...],...\}$, which is a hash-index with scalars for keys and lists of tuples for values.

3.2.3 Scopes & Name Binding. The scoping rules of the layout algebra are somewhat more complex than the relational algebra. There are two ways to bind a name in the layout algebra: by creating a relation or by using an operator which creates a scope.

All of the operators in the layout algebra return a relation. Some operators simply pass through the names in their parameter relations. Others, such as `select` and **`scalar`** can be used for renaming or for creating new fields.

Some operators, such as `depjoin`, create a *scope*. A scope is a tag which uniquely identifies the binding site of a name. For example, in `depjoin`$(q\ \texttt{as}\ s, q')$, a field f from q is bound in q' as $s.f$. Scoped names with distinct scopes are distinct and scoped names are distinct from unscoped names. We add scopes to the layout algebra as a syntactically lightweight mechanism for renaming an entire relation. Renaming entire relations is necessary because shadowing is prohibited in the layout algebra. Prohibiting shadowing removes a major source of complexity when writing transformations. While we could use `select` for renaming, we opted to add scopes so that renaming at binding sites would be part of the language rather than a pervasive and verbose pattern.

There are still situations when renaming entire relations using `select` is necessary. For example, in a self-join one side of the join must be renamed.

3.3 Preview of Layout & Runtime Semantics

In this section, we give a preview of the layout and runtime semantics, which are discussed in detail in Section 6.1 and Section 6.3. The layout semantics specifies the layout of data in memory at a byte level. Each layout operator has a serialization format, and the semantics describes how these formats are composed together. The runtime semantics describes how the layout and query operators read the data from that serialization format and produce a query output.

The nesting and ordering of the layout operators in a query correspond directly to the nesting and ordering of the data structures that they represent. This means that we can reorder or transform operators in the query to restructure the layout. Castor supports the following data structures, each of which has a corresponding layout operator:

- **Scalars:** Scalars can be integers, strings, Booleans, and decimal fixed-points.
- **Tuples:** Tuples are layouts that contain layouts of different types. If a collection contains tuples, all the tuples must have the same number of elements and their elements must have the same type. Tuples can be read either by taking the cross product or concatenating their sub-layouts.
- **Lists:** Lists are variable-length layouts. Their contents must be of the same type.
- **Hash indexes:** Hash indexes are mappings between scalar keys and layouts, stored as hash tables. Like lists, their keys must have the same type and their values must have the same type.
- **Ordered indexes:** Ordered indexes are ordered mappings between scalar keys and layouts.

At runtime, the layout operators read data from the layout and convert it into a relational form that the relational operators can consume. In Section 6.3, we discuss how these operators are implemented as iterators and how the iterators are composed together to form an executable query.

3.4 Staging

Another way to view the three semantic interpretations is from the point of view of multi-stage programming: the layout is constructed in the compile-time stage and the compiled query reads the layout and processes it in the run-time stage. While traditionally program staging is used to implement code specialization, in the layout algebra staging is used to implement data specialization. This difference in focus leads to different implementation challenges. In particular, the “unstaged” version of a layout algebra program is often large (tens to hundreds of megabytes). The Castor compiler must be carefully designed to handle this scale.

We are particularly interested in layout algebra programs that can be separated into a compile time stage that constructs a layout and a runtime stage that reads it. Only a subset of layout algebra programs can be separated in this way. We say that programs which have this property are *well-staged*.

At a high level, a program is well-staged if there are no compile-time dependencies on run-time data or vice versa. To formalize this intuition, we introduce run- and compile-time contexts. An expression is in a compile-time context if it appears in the first argument to **`list`**, **`hash-idx`**, **`ordered-idx`**, or **`scalar`**. Otherwise, it is in a run-time context. Additionally, the fields of relations are considered compile-time only and query parameters are run-time only. A program is well-staged if and only if the names referred to in compile-time contexts are bound in compile-time contexts and the names

$$S ::= \{x_1 \mapsto e_1,\ldots,x_m \mapsto e_m\} \quad T ::= [C,\ldots,q_n] \mid \ldots \mid [q_1,\ldots,C]$$
$$\begin{aligned}C ::= &\ [\cdot] \mid \texttt{select}(S, C) \mid \texttt{filter}(e, C) \mid \texttt{join}(e, q, C) \mid \texttt{group-by}(S, E, C) \mid \texttt{dedup}(C)\\ \mid &\ \textbf{list}(q \text{ as } x, C) \mid \textbf{list}(C \text{ as } x, q) \mid \textbf{tuple}_\tau(T) \mid \textbf{hash-idx}(C \text{ as } x, q_v, e_k) \mid \textbf{hash-idx}(q_k \text{ as } x, C, e_k)\\ \mid &\ \textbf{ordered-idx}(C \text{ as } x, q_v, e_{lo}, e_{hi}) \mid \textbf{ordered-idx}(q_k \text{ as } x, C, e_{lo}, e_{hi})\end{aligned}$$

Fig. 7. The grammar of contexts.

referred to in run-time contexts are bound in run-time contexts. The compiler uses a simple type system that tracks the stage of each name in the program to check well-stagedness.

Transforming a program into a well-staged form is a key goal of our automatic optimizer (Section 5). Many of the rules that the optimizer applies can be seen as moving parts of the query between stages.

4 TRANSFORMATIONS

In this section, we define semantics preserving transformation rules that optimize query and layout performance. These rules change the behavior of the program with respect to the layout and runtime semantics while preserving it with respect to the relational semantics. These rules subsume standard query optimizations because in addition to changing the structure of the query, they can also change the structure of the data that the query processes.

4.1 Notation

Transformations are written as inference rules. When writing inference rules, e will refer to scalar expressions and q will refer to layout algebra expressions. E and Q will refer to lists of expressions and layouts. In general, the names we use correspond to those used in the syntax description (Figure 5). If we need to refer to a piece of concrete syntax, it will be formatted as e.g., `concat` or `x`.

To avoid writing many trivial inductive rules, we define contexts (Figure 7) [Felleisen and Hieb 1992]. If C is a context and q is a layout algebra expression, then $C[q]$ is the expression obtained by substituting q into the hole in C. In addition to contexts, we define two operators: $\xrightarrow{t}$ and $\rightarrow$. $q \xrightarrow{t} q'$ means that the layout algebra expression q can be transformed into q' and $q \rightarrow q'$ means that q can be transformed into q' in any context. The relationship between these two operators is:

$$q \rightarrow q' \equiv \forall C.\ C[q] \xrightarrow{t} C[q']$$

4.2 Relational Optimization

There is a broad class of query transformations that have been developed in the query optimization literature [Chaudhuri 1998; Jarke and Koch 1984]. These transformations can generally be applied directly in Castor, at least to the relational operators. For example, commuting and reassociating joins, filter pushing and hoisting, and splitting and merging filter and join predicates are implemented in Castor. Although producing optimal relational algebra implementations of a query is explicitly a non-goal of Castor, these kinds of transformations are important for exposing layout optimizations.

4.3 Projection

Projection, or the removal of unnecessary fields from a query, is an important transformation because many queries only use a small number of fields; the most impactful layout specialization that can be performed for these queries is to remove unneeded fields.

First, we need to decide what fields are necessary. For a query q in some context C, the necessary fields in q are visible in the output of $C[q]$ or are referred to in C. Let $\textsc{free}(\cdot)$ be a function which returns the set of free variables in a context or layout expression. Let $\textsc{needed}(\cdot,\cdot)$ be a function from contexts C and layouts q to the set of necessary fields in the output of q:

$$\textsc{needed}(C, q) = \textsc{schema}(q) \cap (\textsc{schema}(C[q]) \cup \textsc{free}(C))$$

$\textsc{needed}(\cdot,\cdot)$ can be used to define transformations which remove unnecessary parts of a layout. For example, this rule removes unnecessary fields from tuples:

$$\frac{Q' = [q \mid q \in Q, \textsc{needed}(C, q) \neq \emptyset]}{C[\mathbf{tuple}_\tau(Q)] \xrightarrow{t} C[\mathbf{tuple}_\tau(Q')]}$$

There is a similar rule for `select` and `group-by` operators.

The projection rules differ from the others in this section because they refer to the context C. The other rules can be applied in any context. The context is important for the projection rules because without it, all the fields in a layout would be visible and therefore "necessary". Referring to the context allows us to determine which fields are visible to the user.

4.4 Precomputation

A simple transformation that can improve query performance is to compute and store the values of parameter-free terms. This transformation is similar to partial evaluation. The following rule[4] precomputes a static layout algebra expression:

$$\frac{\textsc{schema}(\boxed{q}) = [f_1, \ldots, f_k] \qquad x \text{ is fresh}}{\boxed{q} \to \mathbf{list}(\boxed{q} \;\mathtt{as}\; x, \mathbf{tuple}_{\text{cross}}([\mathbf{scalar}(x.f_1), \ldots, \mathbf{scalar}(x.f_k)]))}$$

Hoisting static expressions out of predicates can also be very profitable:

$$\frac{x, y \text{ are fresh} \qquad \boxed{e'} \text{ is a term in } e \qquad \textsc{free}(e') \cap \textsc{schema}(q) = \emptyset}{\mathtt{filter}(e, q) \to \mathtt{depjoin}(\mathbf{scalar}(e' \mapsto y) \;\mathtt{as}\; x, \mathtt{filter}(e[e' := x.y], q))}.$$

The expression e' can be precomputed and stored instead of being recomputed for every invocation of the filter. Similar transformations can be applied to any operator that contains an expression. This rule is useful when the filter appears inside a layout operator. For example, in the query $\mathbf{list}(q \;\mathtt{as}\; x, \mathtt{filter}(e, q'))$, a sub-expression of e can be hoisted out of the filter if it refers to the fields in q but not if it refers to the fields in q'.

In a similar vein, `select` operators can be partially precomputed. For example:

$$\frac{y' \text{ is fresh} \qquad q'_v = \mathtt{select}(\{\mathtt{sum}(e) \mapsto y'\}, q_v)}{\begin{array}{c}\mathtt{select}(\{\mathtt{sum}(e) \mapsto y\}, \mathbf{ordered\text{-}idx}(q_k \;\mathtt{as}\; x, q_v, t_{lo}, t_{hi})) \to \\ \mathtt{select}(\{\mathtt{sum}(y') \mapsto y\}, \mathbf{ordered\text{-}idx}(q_k \;\mathtt{as}\; x, q'_v, t_{lo}, t_{hi}))\end{array}}.$$

After this transformation, the ordered index will contain partial sums which will be aggregated by the outer select. This rule is particularly useful when implementing grouping and filtering queries, because the filter can be replaced by an index and the aggregate applied to the contents of the index. A similar rule also applies to `select` and **`list`**. A simple version of this rule applies to **`hash-idx`**; in that case, the outer `select` is unnecessary.

This transformation is combined with group-by elimination (Section 4.5) in TPC-H query 1 to construct a layout that precomputes most of the aggregation.

[4]Some of the rules make a distinction for *parameter-free* expressions, which do not contain query parameters. In these rules, parameter-free expressions are denoted as $\boxed{e}$.

4.5 Partitioning

Partitioning is a fundamental layout transformation that splits one layout into many layouts based on the value of a field or expression. A partition of a relation r is defined by an expression e over the fields in r. Tuples in r are in the same partition if and only if evaluating e over their fields gives the same value.

To simplify the rules involving partitions, we define a function $\textsc{part}(\cdot,\cdot,\cdot)$ which takes a layout q, a partition expression e, and a name x, and returns a pair of queries q_k and q_v:

$$\textsc{part}(q, \boxed{e}, x) = (q_k, q_v) = (\texttt{dedup}(\texttt{select}(\boxed{e}, q)), \texttt{filter}(x.e = \boxed{e}, q)).$$

In this definition, q_k evaluates to the unique valuations of e in r. These are the partition keys. Note that the expression q_v contains a free scope x. We use $x.e$ to denote the expression e with its names qualified by the scope x. Once $x.e$ is bound to a particular partition key, q_v evaluates to a relation containing only tuples in that partition.

The partition function is used to define rules that create hash and ordered indexes from filters:

$$\frac{x,n \text{ is fresh} \qquad \textsc{part}(q, \boxed{e}, x) = (q_k, q_v) \qquad \textsc{free}(e') \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(\boxed{e} = e', q) \rightarrow \mathbf{\texttt{hash-idx}}(q_k \text{ as } x, q_v, e')} \text{ and } \frac{x \text{ is fresh} \qquad \textsc{part}(q, \boxed{e}, x) = (q_k, q_v) \qquad (\textsc{free}(e_l) \cup \textsc{free}(e_h)) \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(e_l \leq \boxed{e} \wedge \boxed{e} \leq e_h, q) \rightarrow \mathbf{\texttt{ordered-idx}}(q_k \text{ as } x, q_v, e_l, e_h)}.$$

Partitioning also leads immediately to a rule that eliminates $\texttt{group-by}(\cdot)$:

$$\frac{x \text{ is fresh} \qquad \textsc{part}(q, \boxed{K}, x) = (q_k, q_v)}{\texttt{group-by}(E, \boxed{K}, q) \rightarrow \mathbf{\texttt{list}}(q_k \text{ as } x, \texttt{select}(E, q_v))}.$$

There is a slight abuse of notation in this rule. K is a list of expressions, so the filter in q_v must have an equality check for each expression in K. This group-by elimination rule is used in many of the TPC-H queries which contain `group-bys`.

4.6 Join Elimination

Partitioning can be used to implement join materialization: a powerful transformation that can significantly reduce the computation required to run a query, at the cost of increasing the size of the data that the query runs on. Joins are often the most expensive operations in a relational query, so choosing a good join materialization strategy is critical. Castor's layout operators admit several options for join materialization.

For example, joins can be materialized as a list of pairs:

$$\frac{x \text{ is fresh} \qquad \textsc{part}(q, \boxed{e}, x) = (q_k, q_v) \qquad \textsc{part}(q', \boxed{e'}, x) = (q'_k, q'_v) \qquad \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad \textsc{free}(e') \subseteq \textsc{schema}(q')}{\texttt{join}(\boxed{e} = \boxed{e'}, q, q') \rightarrow \mathbf{\texttt{list}}(\texttt{join}(e = e', q_k, q'_k) \text{ as } x, \mathbf{\texttt{tuple}}_{\text{cross}}([q_v, q'_v]))}.$$

Each pair in this layout contains the tuples that should join together from the left- and right-hand-sides of the join.

Joins can also be materialized as nested lists:

$$\frac{x \text{ is fresh} \quad \textsc{schema}(q) = [f_1, \ldots, f_n] \quad F = \mathbf{\texttt{scalar}}(f_1), \ldots, \mathbf{\texttt{scalar}}(f_n) \qquad \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad \textsc{free}(e') \subseteq \textsc{schema}(q')}{\texttt{join}(e = e', \boxed{q}, q') \rightarrow \mathbf{\texttt{list}}(\boxed{q} \text{ as } x, \mathbf{\texttt{tuple}}_{\text{cross}}([F, \texttt{filter}(x.e = e', q')]))}.$$

This layout works well for one-to-many joins, because it only stores each row from the left hand side of the join once, regardless of the number of matching rows on the right hand side.

Or, joins can be materialized using a hash table:

$$\frac{\begin{array}{c} x{,}x' \text{ are fresh} \quad \textsc{part}(q', \boxed{e'}, x') = (q_k, q_v) \quad \textsc{schema}(q) = [f_1,\ldots,f_n] \quad \textsc{schema}(q') = [f'_1,\ldots,f'_m] \\ \textsc{free}(e) \subseteq \textsc{schema}(q) \qquad\qquad \textsc{free}(e') \subseteq \textsc{schema}(q') \end{array}}{\texttt{join}(e = \boxed{e'}, q, q') \rightarrow \texttt{depjoin}(q \texttt{ as } x, \texttt{select}([f_1,\ldots,f_n,f'_1,\ldots,f'_m], \textbf{hash-idx}(q_k \texttt{ as } x', q_k, q_v, x.e)))}.$$

This is similar to how a traditional database would implement a hash join, but in our case the hash table is precomputed. Using a hash table adds some overhead from the indirection and the hash function but avoids materializing the cross product if the join result is large.

If the join is many-to-many with an intermediate table, then either of the above one-to-many strategies can be applied.

4.7 Predicate Precomputation

In some queries, it is known in advance that a parameter will come from a restricted domain. If this parameter is used as part of a filter or join predicate, precomputing the result of running the predicate for the known parameter space can be profitable, particularly when the predicate is expensive to compute. Let p be a query parameter and D_p be the domain of values that p can assume.

$$\frac{\begin{array}{cc} \textsc{params}(e) = \{p\} & w_i = e[p := D_p[i]] \\ e' = \bigvee_i (w_i \wedge p = D_p[i]) \vee e & q' = \texttt{select}([w_1,\ldots,w_{|D_p|},\ldots], q) \end{array}}{\texttt{filter}(e, q) \rightarrow \texttt{filter}(e', q')}$$

This rule generates an expression w_i for each instantiation of the predicate with a value from D_p. The w_is are selected along with the original query q. When we later create a layout for q, the w_is will be stored alongside it. When the filter is executed, if the parameter p is in D_p, the or will short-circuit and the original predicate will not run. However, this transformation is semantics preserving even if D_p is underapproximate. If the query receives an unexpected parameter, then it executes the original predicate e. Note that in the revised predicate e', $p = D_P[i]$ can be computed once for each i, rather than once per invocation of the filter predicate.

We use this transformation on TPC-H queries 2 and 9 to eliminate expensive string comparisons.

4.8 Correctness

To show that the semantics that we have outlined in Section 3.2 are sufficient to prove the correctness of nontrivial transformations, we prove the correctness of the equality filter elimination rule (Section 4.5) in Appendix A. Although we do not prove the correctness of all of the rules, this example demonstrates that such proofs are possible.

In particular, since our notation mixes relational and layout constructs, even transformations that manipulate both the run- and compile-time behavior of the query are often local transformations, and are therefore simple to prove correct.

5 OPTIMIZATION

Castor includes an automatic, cost guided optimizer for the layout algebra. The goal of the optimizer is to produce a *transformation sequence*, which is a sequence of transformations that (1) makes the query well-staged (Section 3.4) and (2) minimizes the cost of executing the query. The optimizer consists of two components: a transformation scheduling language and a cost model for the layout algebra.

5.1 Scheduling

The space of transformation sequences is far too large for an exhaustive search. Instead, we consider a restricted space of sequences. We implemented a small domain specific language—the *scheduling* *language*—that describes a search space of transformation sequences. This language is inspired by Stratego [Visser 2005] and provides combinators for sequencing transformations, fix-points, selecting locations to apply transformations, and branching. Running a program in the scheduling language performs a search over transformation sequences. The optimizer is implemented as a program in the scheduling language, and it captures some of the domain knowledge that we have about how to optimize query layouts.

The optimizer scheduling program has four phases: join nest elimination, hash-index introduction, ordered-index introduction, and precomputation. Cleanup transformations and other manipulations that allow the main transformations to apply are interleaved between these phases.

The join nest elimination phase looks for unparameterized join nests and replaces them with layouts. As discussed in Section 4.6, there are several ways to eliminate a join operator. The right choice depends on whether the join is one-to-one or one-to-many. To eliminate a join nest, the optimizer performs an exhaustive search using the join elimination rules and uses the cost model to choose the least expensive candidate.

The hash- and ordered-index introduction phases attempt to replace filter operators with indexes. When replacing a filter operator with an index, the most important choice to make is where in the query to place the filter. This choice determines which part of the layout the index will partition. The optimizer exhaustively searches over the possible index placements and uses the cost model to select the best candidate.

Finally the precomputation phase selects unparameterized parts of the query to be computed and stored.

Before returning the query, the scheduler checks that it is well-staged (Section 3.4). If it is not, the query is discarded and scheduling fails.

We run the optimizer scheduling program in a Markov chain Monte Carlo outer loop that randomly disables transformations. A single run of the optimizer consists of many runs of the scheduling program, with transformations randomly disabled. We use the cost model to decide when to transition in the Markov chain. We keep track of the best transformation sequence that we have found and return that at the end of optimization.

The output of the optimizer is a sequence of transformation rules that introduce layout operators, minimizing the cost of executing the resulting query. A pleasant feature of the optimizer is that because it simply runs transformation rules, it is semantics preserving if all of the rules are. This means that all sequences are equally correct—they differ only in the quality of their result.

Manual Optimization. The scheduling language can also be used to write manual transformation scripts. The optimizer scheduling program is general purpose and includes several rounds of backtracking search, but the scheduling language can easily represent straight-line sequences of transformations.

5.2 Cost Model

When optimizing a query, we care primarily about its runtime cost; we assume that any compile time cost is acceptable. The staged nature of the layout algebra makes estimating the runtime cost of a query complicated because the runtime cost depends on the sizes of the data structures in the layout. To compute these sizes we need to execute the compile-time portion of the query.

To assist in this cost estimation we introduce an abstraction of the query that we call a *layout shape* (Figure 8). A layout shape is essentially an abstract domain for the layout portion of a query. We use an interval abstraction to track the range of integer and fixed-point values in the layout as well as the number of elements in collections like lists and indexes. Given a layout shape, we can we use simple models of the costs of the runtime query operators to estimate the cost of executing the entire query.

$$n ::= \mathbb{Z} \quad r ::= [n, n]$$
$$t ::= \mathsf{intT}(r) \mid \mathsf{boolT} \mid \mathsf{fixedT}(r, n_{scale}) \mid \mathsf{stringT}(r_{chars}) \mid \mathsf{tupleT}([t_1, ..., t_k]) \mid \mathsf{listT}(t, r_{elems})$$
$$\mid \ \mathsf{hash\text{-}idxT}(t_k, t_v, r_{keys}) \mid \mathsf{ordered\text{-}idxT}(t_k, t_v, r_{keys}) \mid \mathsf{emptyT} \mid \mathsf{funcT}(t_1,...,t_m)$$

Fig. 8. The syntax of layout shapes.

$$\frac{\sigma \vdash \mathbf{scalar}(e \mapsto n) \Downarrow x \quad x \text{ is an integer}}{\sigma \vdash \mathbf{scalar}(e \mapsto n) : \mathsf{intT}([x,x])} \quad \frac{\sigma \vdash q_1 : t_1, ..., \sigma \vdash q_k : t_k \quad t = \mathsf{tupleT}([t_1,...,t_k], \tau)}{\sigma \vdash \mathbf{tuple}_\tau([q_1,...,q_k]) : t} \quad \frac{\sigma \vdash q : t}{\sigma \vdash \mathtt{filter}(e, q) : \mathsf{funcT}(t)}$$

$$\frac{\sigma \vdash q_k \Downarrow r \qquad t_v = \bigsqcup_{\sigma' \in r, \sigma' \vdash q_v : t'} t'}{\mathbf{list}(q_k \text{ as } n, q_v) : \mathsf{listT}(t_v, [|r|, |r|])} \quad \frac{t = \mathsf{intT}([l, h]) \qquad t' = \mathsf{intT}([l', h'])}{t \sqcup t' = \mathsf{intT}([\min(l, l'), \max(h, h')])}$$

Fig. 9. Selected semantics of the shape inference pass.

Computing the layout shape is expensive because it involves running the compile time portion of the query. We want to compute the costs of many layouts during optimization, so optimizing the shape computation is critical. We use two techniques during optimization to make the shape computation cheaper. First, we compute the shape on a sample of the database. Using a sample means that the shape may be underapproximate (the ranges in the sample shape will be smaller than in the true shape), but we have found that this is acceptable during optimization. Second, we compute the shape of nested layouts in parallel. For example, to compute the shape of this layout:

$$\mathbf{list}(\mathtt{dedup}(\mathtt{select}(\{id\}, log)) \text{ as } a, \mathbf{list}(\mathtt{filter}(a.id = id, log) \text{ as } b, \mathbf{scalar}(b.exit))),$$

Castor will issue these SQL queries:

```
select count(distinct id) as x1      select min(c) as x2, max(c) as x3      select min(exit) as x4,
from log;                            from (select count() as c                     max(exit) as x5
                                          from log group by id);            from log;
```

It uses the results to construct this shape: $\mathsf{listT}(\mathsf{listT}(\mathsf{intT}([x_4, x_5]), [x_2, x_3]), [x_1, x_1])$, where $[x_4, x_5]$ is the domain of the values of the `scalar` and $[x_2, x_3]$ and $[x_1, x_1]$ are the domains of the **list** lengths. These queries can be run concurrently, and if one of the queries times out we can approximate its results while still being able to compute the rest of the shape.

6 COMPILATION

The result of running the optimizer or manually applying transformation rules is a program in the layout algebra. This program is still quite declarative, so there is a significant abstraction gap to cross before the program can be executed efficiently. Compilation of layout algebra programs proceeds in three phases: data structure specialization, serialization and code generation.

6.1 Layout Semantics

The layout semantics describe how the layout portion of the program is converted to a binary representation. We use the specialized data structure (Section 6.2) for each operator to determine exactly how the layout is serialized.

Each of the layout operators has a binary serialization format which is intended to (1) take up minimal space and (2) minimize the use of pointers to preserve data locality.

- Integers are stored using a variable number of bytes (1-8).
- Dates are represented as the number of days since the epoch and stored as integers.
- Booleans are represented as integers that are either 0 or 1.

$$b : \mathit{Byte\ string} \quad \sigma, t : \mathit{Tuple} \quad \delta : \mathit{Id} \mapsto \mathit{Relation}$$

$$\frac{}{\sigma,\delta \vdash \emptyset \downarrow \text{""}} \qquad \frac{\sigma, \delta \vdash q \downarrow b}{\sigma, \delta \vdash \texttt{filter}(e, q) \downarrow b} \qquad \frac{\sigma, \delta \vdash q \downarrow b \quad \sigma, \delta \vdash q \downarrow b'}{\sigma, \delta \vdash \texttt{join}(e, q, q') \downarrow bb'} \qquad \frac{\sigma, \delta \vdash e \Downarrow v \quad b \text{ is the binary format of } v}{\sigma, \delta \vdash \textbf{scalar}(e) \downarrow b}$$

$$\frac{\sigma, \delta \vdash q_k \Downarrow [t_1,\ldots,t_n] \quad \forall 1 \le i \le n.\ \sigma \cup t_i, \delta \vdash q_v \downarrow b_i \quad \sigma, \delta \vdash \textbf{scalar}(|b_1|+\cdots+|b_n|) \downarrow b_{len} \quad \sigma, \delta \vdash \textbf{scalar}(n) \downarrow b_{ct}}{\sigma, \delta \vdash \textbf{list}(q_k, q_v) \downarrow b_{ct} b_{len} b_1 \ldots b_n} \qquad \frac{\forall 1 \le i \le n.\ \sigma, \delta \vdash q_i \downarrow b_i \quad \sigma, \delta \vdash \textbf{scalar}(|b_1|+\cdots+|b_n|) \downarrow b_{len}}{\sigma, \delta \vdash \textbf{tuple}_\tau([q_1,\ldots,q_n]) \downarrow b_{len} b_1 \ldots b_n}$$

Fig. 10. Selected layout semantics.

- Fixed point numbers are normalized to a fixed scale, and stored as integers.
- Strings are length-prefixed and are not null terminated.
- Tuples are length-prefixed concatenations of their child layouts.
- Lists are stored as a length and an element count followed by the concatenation of their elements. They can be efficiently scanned through, but not accessed randomly by index, because their elements may be variable-sized.
- Hash indexes are implemented using minimal perfect hashes that index into a table of value offsets.
- Ordered indexes contain an ordered table of keys and value offsets. Lookups are performed using a binary search on this table.

Serialization proceeds as described in Figure 10. Each layout operator is serialized by first serializing its children, then constructing the appropriate data structure. For example, to construct the layout for $\textbf{list}(q_k \text{ as } k, q_v)$, we first serialize the dependent relation q_v for each tuple in q_k. We concatenate the resulting layouts and prepend the header fields *count* and *length*, which contain the number of list items and the length of the list in bytes respectively. The query operators have trivial layout semantics that simply concatenate their child layouts.

6.2 Data Structure Specialization

The first step in the Castor compiler is to generate specialized instances of the layout algebra data structures. Each instance of a layout operator in a query gets a specialized data structure implementation. We use the shape (Section 5.2) of the layout to guide the specialization process.

First, we compute the layout shape of the query. Unlike the optimizer, which uses a fast underapproximate method to compute the shape, the compiler needs an overapproximation. Computing an overapproximate shape ensures that if we construct a data structure that supports the values in the shape, it will support all of the values in the true dataset.

Our data structure implementations follow a common pattern. They consist of a header that contains fields like lengths, counts, or offsets and a body that contains the data. For example, strings are represented using a length header and a body that contains the byte-encoded data. Castor supports the following generic specializations for all structures:

- *Field narrowing:* Castor uses the layout shape to determine the range of certain header fields (like string lengths). It then chooses the smallest byte-width that is large enough to support the range of values in the field.
- *Field elision:* If the layout shape shows that a field will only ever contain a single value, then Castor elides the field entirely. For example, this specialization allows fixed-size tuples to be stored with zero overhead.

Castor implements additional specializations for certain layout operators.

Integers are narrowed in the same way that fields are. We use the layout shape to determine the minimal number of bytes necessary to hold the values that appear in the layout. The narrowing can differ for each integer layout operator.

For fixed point numbers, the layout shape contains a minimal scaling factor that is precise enough for all of the values and a domain covering the numerators of the fixed points. The numerators are stored as integers and the shared scaling factor is not stored. Each fixed point layout operator can have a different scaling factor.

Hash indexes are implemented using a minimal perfect hash that is computed per-index [Botelho et al. 2007; Davi de Castro Reis et al. 2011]. Using a minimal perfect hash allows our hash indexes to have high load factors (up to 99%) and also allows us to ignore the possibility of collisions.

After performing data structure specialization, we serialize the layout as described in Section 6.1, using the specializations to decide which fields to narrow or remove.

6.3 Runtime Semantics

The last step in compilation is to generate the code that reads the layout. Each operator has a corresponding iterator: the layout operators read from the layout and produce a stream of tuples and the relational operators consume and produce tuple streams. We construct these iterators according to the method in [Tahboub et al. 2018]. This method is referred to as push-based or data-centric query evaluation.

In push-based evaluation, each iterator takes a callback which it calls for each tuple in its output stream. To run the query, the user passes the root iterator a callback function that processes the output tuples. For a given query, the callbacks are known statically, so we inline them. This produces a single loop nest with no function calls and minimal branching.

Each operator in the layout algebra has a corresponding iterator implementation. The relational operators are implemented as described in [Tahboub et al. 2018]. The layout operators each have an iterator that reads the layout. These iterators are modified depending on the data structure specializations being performed. Specifically, the iterators can be modified to implement the field narrowing and elision specializations discussed in Section 6.2.

The push-based evaluation method is in contrast to pull-based evaluation, which is used in the traditional iterator model. In pull-based evaluation, each query operator is compiled to an iterator that consists of a state structure and a step function. Each time a query operator is stepped, it recursively steps its children and updates its state. The query is executed by repeatedly stepping the root iterator. We implemented pull-based evaluation in an early prototype of Castor and found that optimizing the resulting code was difficult because of the large amount of branching and control flow.

The drawback of push-based query evaluation is that certain operators, such as deduplication and ordering, must buffer their inputs before processing them. Rather than implement buffering, we restrict the use of these operators and wherever possible we replace them with layout-based implementations that perform these operations at compile time.

Code Generation. Castor performs code generation in two phases. First, a syntax-directed lowering pass transforms each query and layout operator into an imperative intermediate representation, using the layout shape to generate the layout reading code. Next, we run loop invariant code motion and reorder the evaluation of predicates so that expensive clauses in conjunctive predicates are evaluated last. Finally, we lower the imperative IR to LLVM IR. LLVM performs further low-level optimizations and emits code, completing the code generation phase.

6.4 Performance of Staging

As discussed in Section 5.2, running the compile time part of a query is costly. During compilation of a query, we run its compile time part once to generate a layout shape and again during serialization. We evaluate the compile time part of a query by translating it to a SQL query, running that query against a backing database, then folding over the results. This process is the most expensive part of compilation, since it involves executing a complex query and reading a large amount of data from the database. Unlike the optimizer, the compiler cannot tolerate an approximate result, so we implemented two optimizations that do not involve approximation to manage this cost.

First, we perform a series of query optimizations before sending a query to the database. The most important optimization is the removal of dependent joins [Neumann and Kemper 2015]. Each level of layout nesting introduces an additional dependent join, and many database optimizers are not able to optimize them away. Removing them significantly improved the performance of our queries.

Second, we use Amazon Redshift as the backing database for the Castor compiler. Redshift is a high-performance distributed column oriented database, and switching to Redshift from PostgreSQL yielded a significant performance improvement. Castor is not a good fit for this use case, because the queries that the compiler generates are specific to the program being compiled, so are not amenable to caching.

After implementing these optimizations, the compilation time for TPC-H query 3 dropped from over 12 hours to less than 20 minutes. Running the compiler using a 2 node Redshift cluster, the maximum compilation time for an optimizer generated query is 100 minutes with a median of 4.8 minutes.

7 EVALUATION

We test three hypotheses about Castor's performance: (1) Castor's layout optimizations produce queries that are faster than existing in-memory databases, (2) the resulting layouts are smaller than in existing databases, and (3) the deductive approach to query optimization scales better than generate-and-test approaches.

We compare Castor with three other systems: Hyper [Neumann 2011], Cozy [Loncaric et al. 2018], and Chestnut [Yan and Cheung 2019]. Hyper is an in-memory column-store which has a state-of-the art vectorizing query compiler. It implements compilation techniques that are well outside the scope of this paper, such as using SIMD operations to operate on multiple tuples at a time. Cozy is a state-of-the-art generate-and-test based program synthesis tool that generates specialized data structures from relational queries. Chestnut is similar to Cozy, but focuses on object queries.

7.1 TPC-H Analytics Benchmark

In Section 2, we evaluated Castor on a query from the DemoMatch system. In this section, we perform an in-depth evaluation on the TPC-H benchmark. TPC-H is a standard database benchmark, focusing on analytics queries. It consists of a data generator, 22 query templates, and a query generator which instantiates the templates. The queries in TPC-H are inherently parametric, and their parameters come from the domains defined by the query generator. To build our benchmark, we took the query templates from TPC-H and encoded them as Castor programs. It is important that the queries be parametric. Specializing a non-parametric query is uninteresting because it can simply be evaluated and the result stored.

TPC-H is a general purpose benchmark, so it exercises a variety of SQL primitives. We chose not to implement all of these primitives in Castor, not because they would be prohibitively difficult, but because they are not directly related to the layout specialization problem. In particular, Castor does not support executing `order-by`, `group-by`, `join`, or `dedup` operators at runtime[5], and it does not support limit clauses at all. Some of these operators can be replaced by layout specialization, but others cannot. We implemented all of the queries in TPC-H, except for query 13 because it contains an outer join. We removed runtime ordering and limit clauses from seven other queries. When evaluating the TPC-H queries, we used the 1GB scale factor. We ran our benchmarks on an Intel Xeon W–2155 with 64GB of memory.

7.2 Comparison with Hyper

We run each benchmark query using the following configurations:

- *Baseline:* We use Hyper as our baseline. Each query is run on a database containing the full TPC-H dataset.
- *Manual:* We manually implement the transformations that Castor performs in Hyper. We do this by generating a specialized set of materialized views and indexes that replicate the specialized layout the Castor produces. Hyper supports a smaller space of layouts than Castor, so this translation is best-effort. Hyper does not support nested layouts, for example.
- *Expert:* We run Castor using an expert-written transformation sequence for each query which generates an efficient, well-staged version of the query. The advantages over the manual approach are: (1) the transformations are correctness-preserving, so we don't have to worry about introducing bugs while optimizing and (2) we can use the Castor compiler to generate the specialized layout and query code.
- *Optimizer:* We run the Castor optimizer (Section 5) on a direct translation from the SQL implementation of the query to the layout algebra. The optimizer searches over the space of transformation sequences, using its cost model to select the best sequence. We run the optimizer with a 2 hour timeout.

For each query and configuration, we measure its runtime, layout size, and memory footprint.

Runtime. Figure 11 shows the speedup over baseline Hyper for the *Manual*, *Expert*, and *Optimizer* configurations. The *Manual* configuration requires the most work from the user and offers no assurance of correctness. It is faster than the baseline more than half of the time. The *Expert* configuration is faster than the baseline 85% of the time and faster than the *Manual* configuration 60% of the time. The optimizer beats the baseline 80% of the time. These results show that the *Expert* and *Optimizer* configurations offer a compelling performance advantage over the baseline and a compelling user interface advantage over the *Manual* configuration.

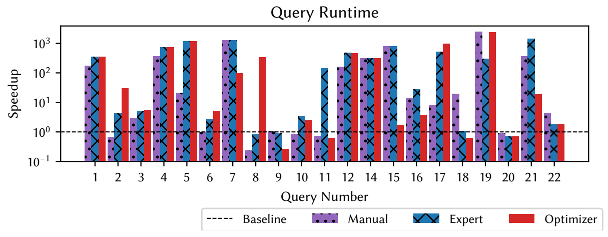


Fig. 11. Performance on TPC-H queries.

Layout Size. We recorded the size of the layouts for the *Manual*, *Expert*, and *Optimizer* configurations. We exclude *Baseline* because it has a large constant size for all queries (approximately the size of the TPC-H data—1GB). Figure 12 shows *Expert* and *Optimizer* with *Manual* as the new baseline. *Expert* beats *Manual* on all benchmarks and *Optimizer* beats *Manual* on 95%.

[5]These operators can be processed into the compiled form of the query.

The absolute size of the *Expert* layouts is small—less than 10MB for 55% of the queries and less than 100MB for 90% of the queries. The size of all layouts is 918.4MB, which compares favorably to the 1.1GB original data set. The size difference between Castor's layouts and the original data supports the hypothesis that parameterized queries rely on fairly small subsets of the whole database, making layout specialization a profitable optimization even when it involves replication.

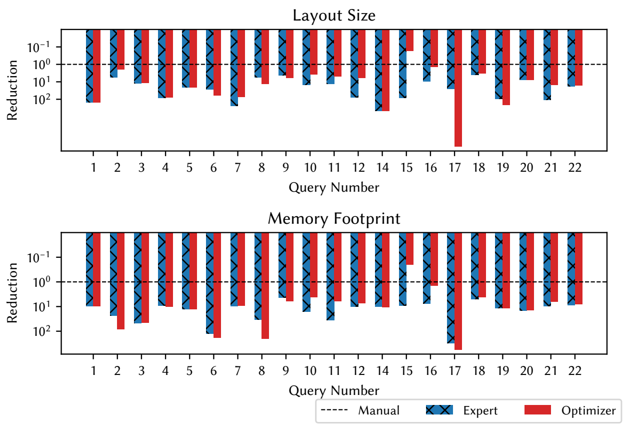


Fig. 12. Layout size of TPC-H queries.

Memory Use. Finally, we measured the peak memory use of the query process for each query. Hyper consistently uses the same amount of memory as the layout size. In some cases it uses more, presumably because it has large runtime dependencies like LLVM. In contrast, Figure 12 shows that Castor's peak memory use is significantly lower than Hyper for all expert queries and for 95% of the optimizer queries.

7.3 Comparison with Cozy & Chestnut

Cozy. We transformed our input queries into Cozy's specification format and ran Cozy with a 6 hour timeout. In this configuration, we found that Cozy was unable to make significant improvements on all but two of the TPC-H queries. On Q4, Cozy precomputed one of the joins and a filter. On Q17, Cozy added an index. We ran both of these queries and found that despite the optimization Q4 was slower than baseline Hyper at 5.4s and Q17 was too slow to run on the entire TPC-H dataset.

There are two reasons why Castor performs better than Cozy in our comparison. First, Castor's deductive approach to optimization scales better than Cozy's generate and test method as the query size increases. This means that Castor is able to spend more time choosing between data structures, rather than looking for a correct implementation. Second, Castor has custom implementations of its data structures that take advantage of the fact that the database is read-only and known to the compiler. Cozy uses the C++ STL collections, which can't make either assumption. So, Castor is somewhat better at choosing layouts and has layouts that are better tuned for its use case.

Chestnut. We attempted to use Chestnut to optimize four of the TPC-H queries, which were implemented as benchmarks by the Chestnut authors, but we were unable to build and run the generated code. [6] Manually examining the layouts for Q1 and Q3–6, we find that Chestnut uses projection and indexes in many of the same places that Castor does, but misses some optimizations that Castor can take advantage of, such as aggregate precomputation (Section 4.4). As Chestnut uses a generate-and-test strategy that is similar to Cozy, it is likely to have similar scalability problems on larger queries. Like Cozy, it relies on C++ collections which do not have data-specific specializations.

[6] At the time of writing, Chestnut was not publicly released, so we used a research prototype that the authors kindly provided to us. We have contacted the Chestnut authors, but have not yet been able to debug the query code.

Table 1. Performance of pairs of queries compiled together.

Query Pair	Query	Runtime (ms)		Memory (MB)		Size (MB)	
		Optimizer	*Manual*	*Optimizer*	*Manual*	*Optimizer*	*Manual*
1 & 2	1	0.02	0.03	2.3	233.9	102.9	206.6
	2	0.08	3.55	2.9	233.9		
2 & 3	2	0.08	3.57	2.9	1088.8	220.6	1088.4
	3	4.77	7.46	25.8	1089.1		
3 & 4	3	5.05	7.20	25.5	879.4	130.8	966.8
	4	0.08	0.02	3.4	879.5		
4 & 5	4	0.08	0.02	3.3	35.4	17.5	21.0
	5	0.02	0.54	2.9	35.5		

7.4 Performance of Multiple Query Workloads

We experimented with combining pairs of queries using the following reduction. A pair of queries q and q' may be reduced to a single query with an additional parameter id that chooses the query to execute:

$$\texttt{filter}(qid = id, \textbf{tuple}_{\text{concat}}([\textbf{tuple}_{\text{cross}}([\textbf{scalar}(0 \mapsto qid), q]), \textbf{tuple}_{\text{cross}}([\textbf{scalar}(1 \mapsto qid), q'])]).$$

We find that the performance of the combined queries is no worse than the performance of the queries when compiled separately. However, our optimizer does not take advantage of sharing opportunities between multiple queries, so the size of the combined layout is the sum of the sizes of the individual layouts. Adding transformations that exploit sharing is left to future work.

7.5 Summary

Our evaluation shows that 95% of the time, Castor produces queries that are faster than a state-of-the-art in-memory database. The layout optimizations that Castor implements are able to outperform a heavily engineered vectorizing compiler, in some cases by multiple orders of magnitude. The layouts produced by Castor are up to two orders of magnitude smaller than a state-of-the-art in-memory database. This reduction in size translates to up to two orders of magnitude reduction in memory footprint. Finally, Castor's deductive optimization scales better than existing generate-and-test synthesis methods. It is able to optimize 21 of the TPC-H queries—more than the existing techniques—while supporting a rich space of optimizations.

8 RELATED WORK

Deductive Synthesis. There is a long line of work that uses deductive synthesis and program transformation rules to optimize programs [Blaine et al. 1998; Püschel et al. 2005], to generate data structure implementations [Delaware et al. 2015], and to build performance DSLs [Ragan-Kelley et al. 2013; Sujeeth et al. 2014]. Castor is a part of this line of work: it is a performance DSL which uses deduction rules to generate and optimize layouts. However its focus on particular data sets and on deduction rules to optimize data in addition to programs separates it from previous work.

Data Representation Synthesis. The layout optimization problem is similar to the problem of synthesizing a data structure that corresponds to a relational specification [Hawkins et al. 2010, 2011; Loncaric et al. 2018, 2016; Sujeeth et al. 2014].

The best data structure synthesis tool—Cozy—uses a generate-and-test strategy. The testing phase uses an SMT solver to perform bounded verification of candidates. In our experiments (Section 7.2) we found that Cozy's verification step does not scale to the TPC-H queries. Castor uses deductive synthesis to avoid this costly verification step by only searching the space of correct programs.

Table 2. Runtime of queries derived from TPC-H (ms). Memory use is the peak resident set size during a query (Mb). Size is the layout size (Mb).

	Manual				*Expert*			*Optimizer*		
Q#	Time[3]	Time[2]	Mem.	Size	Time[2]	Mem.	Size	Time[2]	Mem.	Size
1	5.11	0.03	22.9	17.8	0.01	2.4	0.1	0.01	2.3	0.1
2 [0]	2.34	3.46	231.9	206.6	0.55	9.8	37.2	0.08	2.7	104.0
3 [0,1]	22.35	7.67	877.4	966.8	4.33	18.3	81.0	4.17	19.1	85.6
4	7.33	0.02	22.7	17.8	< 0.01	2.5	0.2	< 0.01	2.2	0.2
5	11.71	0.55	33.3	24.1	< 0.01	2.6	1.2	< 0.01	2.5	1.2
6	2.01	2.10	897.6	858.8	0.74	7.1	31.0	0.41	4.8	14.8
7	12.51	0.01	21.8	17.8	< 0.01	2.3	< 0.1	0.13	2.3	0.2
8	3.46	15.00	527.4	375.4	4.24	15.8	68.9	< 0.01	2.6	29.5
9	35.70	33.79	1544.9	1550.8	39.71	358.6	365.0	132.75	252.7	256.5
10 [0,1]	23.00	28.15	430.3	375.4	6.92	27.1	25.4	8.83	102.9	103.1
11	5.63	7.73	94.7	68.2	0.04	2.6	5.3	8.95	15.7	13.9
12	4.70	0.03	23.1	17.8	< 0.01	2.3	0.2	< 0.01	3.1	3.0
14	3.12	< 0.01	21.8	17.8	< 0.01	2.2	< 0.1	< 0.01	2.1	< 0.1
15	8.06	< 0.01	22.1	17.8	< 0.01	2.4	0.2	4.70	106.8	107.2
16 [1]	39.56	2.78	39.8	35.7	1.41	5.3	3.9	10.94	27.1	25.6
17	9.83	1.20	1224.9	1224.7	0.02	4.1	49.6	< 0.01	2.1	< 0.1
18 [0]	47.66	2.43	375.5	295.7	44.26	74.1	73.5	74.66	88.7	88.5
19	24.44	0.01	25.5	17.8	0.08	2.2	0.2	< 0.01	2.1	< 0.1
20	8.73	9.87	1300.9	1377.8	12.46	91.1	173.3	12.27	91.8	173.3
21 [0,1]	14.33	0.04	22.8	21.0	< 0.01	2.3	0.2	0.75	3.6	1.4
22 [1]	20.09	4.61	33.5	31.5	11.36	3.9	1.8	10.87	4.2	2.0

[0] Limit clause removed. [1] Run time ordering removed. [2] Specialized. [3] Unspecialized.

CHESTNUT is another tool for layout optimization [Yan and Cheung 2019]. It differs from CASTOR and COZY in that it considers object queries from ORMs rather than SQL. Like COZY, CHESTNUT relies on a generate and test strategy with a bounded verification step. Its optimizer exhaustively enumerates layouts and query plans separately and uses the verifier to determine if a plan/layout combination is correct. CHESTNUT avoids the scalability problems inherent to this approach by restricting its search space. For example, it only considers a single level of layout nesting, which limits data locality. It does not consider plans that perform partial aggregation (Section 4.4) or precomputation of predicates (Section 4.7). Any extensions to CHESTNUT must be carefully chosen to avoid search space explosion. In contrast, CASTOR can support complex query transformations in a straightforward way because they do not need to be discovered by an exhaustive search.

Unlike the previous work, CASTOR considers a refinement of the data structure synthesis problem where both the query and the dataset are known to the compiler. This additional information allows CASTOR to use optimizations which would not be safe if the data were not known. In particular, CASTOR is able to significantly reduce the size of its optimized data sets by generating specialized collection implementations based on the properties of the data to be stored. The existing work relies on off-the-shelf collections libraries which cannot be specialized in this way.

Database Storage. Traditional databases are mostly row-based. Column-based database systems (e.g., HYPER [Neumann 2011], MonetDB [Boncz and Kersten 1999] and C-Store [Stonebraker et al. 2005]) are popular for OLAP applications, outperforming row-based approaches by orders of magnitude. However, the existing work on database storage generally considers specific storage optimizations (e.g., [Ailamaki et al. 2001]) or specializations that benefit broad classes of data such as scientific [Stonebraker 2012] or geo-spatial [Gutiérrez and Baumann 2007] data. One exception is RodentStore [Cudré-Mauroux et al. 2009], which proposed a language to describe storage layouts and showed that different layouts could benefit different applications. However, a compiler was never developed to create the layouts from this language; the paper demonstrated its point by implementing each layout by hand.

Materialized View and Index Selection. The layouts that Castor generates are similar to materialized views, in that they store query results. Castor also generates layouts which contain indexes. Several problems related to the use of materialized views and indexes have been studied (see [Halevy 2001] for a survey): (1) the view storage problem that decides which views need to be materialized [Chirkova and Genesereth 2000], (2) the view selection problem that selects view(s) that can answer a given query, (3) the query rewriting problem that rewrites the given query based on the selected view(s) [Pottinger and Levy 2000], (4) the index selection problem that selects an appropriate set of indexes for a query [Bruno and Chaudhuri 2005; Gupta et al. 1997; Stonebraker 1974; Talebi et al. 2008]. However, materialized views are restricted to being flat relations. The layout space that Castor supports is much richer than that supported by materialized views and indexes. In addition, the view selection literature has not previously considered the problem of generating execution plans for chosen views and indexes.

Query Compilation. Castor uses techniques from the query compilation literature [Klonatos et al. 2014; Shaikhha et al. 2016; Tahboub et al. 2018]. It extends these techniques by using information about the layout to specialize its queries.

9 CONCLUSION

We have presented Castor, a domain specific language for expressing a wide variety of physical database designs, and a compiler for this language. We have evaluated it empirically and shown that it is competitive with the state-of-the-art in memory database systems.

A CORRECTNESS OF FILTER ELIMINATION

In this section we discuss the correctness of the filter elimination rule (Section 4.5) in detail. We show that the relational semantics is sufficiently detailed to prove the correctness of the transformation rules.

We say that two programs q and q' are equivalent if they produce the same value in every context. We denote equivalence as $q \equiv q'$ according to the following rule:

$$\text{Equiv}\;\frac{\forall \sigma,\delta,s.\ \sigma,\delta \vdash q \Downarrow s \iff \sigma,\delta \vdash q' \Downarrow s}{q \equiv q'}$$

We say that a rule $q \longrightarrow q'$ is semantics-preserving if $q \equiv q'$.

Now we prove that the filter elimination rule:

$$\frac{x,n \text{ is fresh} \qquad \textsc{part}(q, \boxed{e}, x) = (q_k, q_v) \qquad \textsc{free}(e') \cap \textsc{schema}(q) = \emptyset}{\texttt{filter}(\boxed{e} = e', q) \longrightarrow \texttt{\textbf{hash-idx}}(q_k \texttt{ as } x, q_v, e')}$$

is semantics-preserving.

Theorem A.1. *If* $\textsc{part}(q, e, x) = (q_k, q_v)$ *and* x *is a fresh scope, then*

$$\texttt{filter}(e = e', q) \equiv \texttt{\textbf{hash-idx}}(q_k \texttt{ as } x, q_v, e').$$

Proof. By Equiv, the right-hand-side of this implication is equivalent to:

$$\forall \sigma,\delta,s.\ \sigma,\delta \vdash \texttt{filter}(e = e', q) \Downarrow s \iff \sigma,\delta \vdash \texttt{\textbf{hash-idx}}(q_k \texttt{ as } x, q_v, e') \Downarrow s.$$

By R-HI,

$$\sigma,\delta \vdash \texttt{filter}(e=v, q) \Downarrow s \iff \sigma,\delta \vdash \texttt{depjoin}(q_k\ \texttt{as}\ x, \texttt{filter}(x.e=v, q_v)) \Downarrow s,$$

where $\sigma, \delta \vdash e' \Downarrow v$.

By the definition of PARTITION, $q_k = \texttt{dedup}(\texttt{select}(\{e\}, q))$ and $q_v = \texttt{filter}(x.e=v, q)$, so

$$\sigma,\delta \vdash \texttt{filter}(e=v, q) \Downarrow s \iff$$
$$\sigma,\delta \vdash \texttt{depjoin}(\texttt{dedup}(\texttt{select}(\{e\}, q))\ \texttt{as}\ x, \texttt{filter}(x.e=v, \texttt{filter}(x.e=e, q))) \Downarrow s.$$

We can simplify the filter operators to get:

$$\sigma,\delta \vdash \texttt{filter}(e=v, q) \Downarrow s \iff$$
$$\sigma,\delta \vdash \texttt{depjoin}(\texttt{dedup}(\texttt{select}(\{e\}, q))\ \texttt{as}\ x, \texttt{filter}(x.e=v \land x.e=e, q)) \Downarrow s.$$

Proving the correctness of this simplification is straightforward and does not rely on the correctness of the hash-index introduction rule.

By R-Filter and R-Depjoin (and some abuse of notation), this is equivalent to:

$$[t \mid t \leftarrow \texttt{filter}(e=v, q)] = \left[t \;\middle|\; \begin{array}{c} t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\}, q)) \\ t \leftarrow \texttt{filter}(t'=v \land t'=e, q) \end{array} \right].$$

At this point there are two cases of interest. First, assume that $v \in \texttt{dedup}(\texttt{select}(\{e\}, q))$. By the semantics of dedup, v will appear exactly once in this query result if it appears at all. We can conclude that in this case:

$$\begin{aligned}
&[t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\}, q)), t \leftarrow \texttt{filter}(t'=v \land t'=e, q)] \\
&= [t \mid t'=v, t \leftarrow \texttt{filter}(t'=v \land t'=e, q)] +\!\!+ \\
&\quad [t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\}, q)), t' \neq v, t \leftarrow \texttt{filter}(v=t' \land t'=e, q)] \\
&= [t \mid t \leftarrow \texttt{filter}(v=v \land v=e, q)] +\!\!+ [t \mid t \leftarrow \texttt{filter}(v \neq v \land v \neq e, q)] \\
&= [t \mid t \leftarrow \texttt{filter}(v=e, q)] +\!\!+ [\,] \\
&= [t \mid t \leftarrow \texttt{filter}(v=e, q)].
\end{aligned}$$

In the second case, assume that $v \notin \texttt{dedup}(\texttt{select}(\{e\}, q))$. In this case:

$$\begin{aligned}
&[t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\}, q)), t \leftarrow \texttt{filter}(t'=v \land t'=e, q)] \\
&= [t \mid t' \leftarrow \texttt{dedup}(\texttt{select}(\{e\}, q)), t' \neq v, t \leftarrow \texttt{filter}(v=t' \land t'=e, q)] \\
&= [t \mid t \leftarrow \texttt{filter}(v \neq v \land v \neq e, q)] \\
&= [\,].
\end{aligned}$$

By our assumption, there is no e such that $e=v$, so $\texttt{filter}(e=v, q) = [\,]$.

In both cases, the two programs are equivalent, so we can conclude that the rule is semantics-preserving. □

We can conclude from this proof that showing correctness for the transformation rules is feasible.

B SEMANTICS OF THE LAYOUT ALGEBRA

$$Id = (Scope?, Name) \quad Context = Tuple = \{Id \mapsto Value\} \quad Relation = [Tuple]$$
$$\sigma : Context \quad \delta : Id \mapsto Relation \quad s : Id \quad t : Tuple \quad v : Value \quad r : Relation$$

$$\text{E-Tuple}\ \frac{t = \{n_1 \mapsto e_1, \ldots, n_m \mapsto e_m\} \qquad \forall i.\ \sigma, \delta \vdash e_i \Downarrow v_i}{\sigma, \delta \vdash t \Downarrow \{n_1 \mapsto v_1, \ldots, n_m \mapsto v_m\}} \qquad \text{R-Relation}\ \frac{(x,r) \in \delta}{\sigma, \delta \vdash x \Downarrow r} \qquad \text{R-Empty}\ \frac{}{\sigma,\delta \vdash \emptyset \Downarrow [\,]}$$

$$\text{R-Filter}\ \frac{\sigma,\delta\vdash q\Downarrow r_q \quad r=[t \mid t\leftarrow r_q \quad \sigma\cup t,\delta\vdash e\Downarrow \mathsf{true}]}{\sigma,\delta\vdash \mathtt{filter}(e,q)\Downarrow r}$$

$$\text{R-Depjoin}\ \frac{\sigma,\delta\vdash q\Downarrow r \quad r''=\left[t' \;\middle|\; t\leftarrow r \quad t'\leftarrow r' \quad \sigma\cup t_s,\delta\vdash q'\Downarrow r' \quad t_s=\{s.f\mapsto v \mid (f\mapsto v)\in t\}\right]}{\sigma,\delta\vdash \mathtt{depjoin}(q\ \mathsf{as}\ s,q')\Downarrow r''}$$

$$\text{R-Scalar}\ \frac{\sigma,\delta\vdash e\Downarrow v}{\sigma,\delta\vdash \mathbf{scalar}(n\mapsto e)\Downarrow[\{n\mapsto v\}]} \qquad \text{R-List}\ \frac{\sigma,\delta\vdash \mathtt{depjoin}(q_r\ \mathsf{as}\ s,q)\Downarrow r}{\sigma,\delta\vdash \mathbf{list}(q_r\ \mathsf{as}\ s,q)\Downarrow r}$$

$$\text{R-OrderedIndex}\ \frac{\begin{array}{c}\sigma,\delta\vdash \mathtt{depjoin}(q_k\ \mathsf{as}\ s,\mathtt{filter}(v_{lo}\le s.x\le v_{hi},q_v))\Downarrow r\\ \sigma,\delta\vdash l_{lo}\Downarrow v_{lo} \quad \sigma,\delta\vdash l_{hi}\Downarrow v_{hi} \quad \textsc{schema}(q_k)=[x]\end{array}}{\sigma,\delta\vdash \mathbf{ordered\text{-}idx}(q_k\ \mathsf{as}\ s,q_v,l_{lo},l_{hi})\Downarrow r}$$

$$\text{R-Tuple1}\ \frac{}{\sigma,\delta\vdash \mathbf{tuple}_\tau([\,])\Downarrow[\,]} \qquad \text{R-Tuple2}\ \frac{\sigma,\delta\vdash q_1\Downarrow r_q \quad \sigma,\delta\vdash \mathbf{tuple}_{\mathrm{cross}}([q_2,\ldots,q_n])\Downarrow r_{qs}}{\sigma,\delta\vdash \mathbf{tuple}_{\mathrm{cross}}([q_1,\ldots,q_n])\Downarrow[t\cup ts \mid t\leftarrow r_q \quad ts\leftarrow r_{qs}]}$$

$$\text{R-Tuple3}\ \frac{\sigma,\delta\vdash q_1\Downarrow r_q \quad \sigma,\delta\vdash \mathbf{tuple}_{\mathrm{concat}}([q_2,\ldots,q_n])\Downarrow r_{qs}}{\sigma,\delta\vdash \mathbf{tuple}_{\mathrm{concat}}([q_1,\ldots,q_n])\Downarrow r_q{+}{+}r_{qs}}$$

$$\text{R-HashIdx}\ \frac{\sigma,\delta\vdash \mathtt{depjoin}(q_k\ \mathsf{as}\ s,\mathtt{filter}(s.x=v,q_v))\Downarrow r \quad \sigma,\delta\vdash l\Downarrow v \quad \textsc{schema}(q_k)=[x]}{\sigma,\delta\vdash \mathbf{hash\text{-}idx}(q_k\ \mathsf{as}\ s,q_v,l)\Downarrow r}$$

$$\text{R-Join}\ \frac{\sigma,\delta\vdash q\Downarrow r \quad \sigma,\delta\vdash q'\Downarrow r' \quad s=[t\cup t' \mid t\leftarrow r \quad t'\leftarrow r' \quad \sigma\cup t\cup t',\delta\vdash e\Downarrow \mathsf{true}]}{\sigma,\delta\vdash \mathtt{join}(e,q,q')\Downarrow r}$$

$$\text{R-OrderBy}\ \frac{\sigma,\delta\vdash q\Downarrow r \quad r'\text{ is a permutation of }r \quad r'\text{ is ordered according to the values of }e_1,\ldots,e_n}{\sigma,\delta\vdash \mathtt{order\text{-}by}([e_1o_1,\ldots,e_no_n],q)\Downarrow r}$$

$$\text{R-Select}\ \frac{\begin{array}{c}t\text{ contains no aggregates} \quad \sigma,\delta\vdash q\Downarrow r_q\\ r=[t'' \mid t'\leftarrow r_q \quad \sigma\cup t',\delta\vdash t\Downarrow t'']\end{array}}{\sigma,\delta\vdash \mathtt{select}(t,q)\Downarrow r} \qquad \text{R-SelectAgg}\ \frac{\begin{array}{c}t\text{ contains aggregates}\\ \sigma,\delta\vdash \mathtt{group\text{-}by}(t,[\,],q)\Downarrow r\end{array}}{\sigma,\delta\vdash \mathtt{select}(t,q)\Downarrow r}$$

$$\text{R-Dedup}\ \frac{\sigma,\delta\vdash q\Downarrow r_q \quad \forall t\in r.\ t\in r_q \quad \forall t\in r_q.\ \exists i.1\le i\le|r|\wedge r[i]=t\wedge\forall j.\ j=i\vee t\ne r[j]}{\sigma,\delta\vdash \mathtt{dedup}(q)\Downarrow r}$$

$$\text{R-GroupBy1}\ \frac{\sigma,\delta\vdash q\Downarrow r \quad |r|=0}{\sigma,\delta\vdash \mathtt{group\text{-}by}(t,[\,],q)\Downarrow[\,]}$$

$$\text{R-GroupBy2}\ \frac{\sigma,\delta\vdash q\Downarrow r \quad |r|>0}{\sigma,\delta\vdash \mathtt{group\text{-}by}(\{x_1\mapsto e_1,\ldots,x_m\mapsto e_m\},[\,],q)\Downarrow[\{x_1\mapsto agg(e_1,r),\ldots,x_m\mapsto agg(e_m,r)\}]}$$

$$\text{R-GroupBy3}\ \frac{\begin{array}{c}q_k=\mathtt{dedup}(\mathtt{select}(\{y_1,\ldots,y_n\},q)) \quad q_v=\mathtt{group\text{-}by}(E,[\,],\mathtt{filter}(\bigwedge_{i=1}^n y_i=k.y_i,q))\\ \sigma,\delta\vdash \mathtt{depjoin}(q_k\ \mathsf{as}\ k,q_v)\Downarrow r\end{array}}{\sigma,\delta\vdash \mathtt{group\text{-}by}(E,[y_1,\ldots,y_n],q)\Downarrow r}$$

$$agg(e,r)=\begin{cases}|r| & e=\mathtt{count}\\ \min_{\sigma\cup t,\delta\vdash e'\Downarrow w,t\in r} w & e=\mathtt{min}(e')\\ \max_{\sigma\cup t,\delta\vdash e'\Downarrow w,t\in r} w & e=\mathtt{max}(e')\\ \sum_{\sigma\cup t,\delta\vdash e'\Downarrow w,t\in r} w & e=\mathtt{sum}(e')\\ agg(\mathtt{sum}(e'),r)/agg(\mathtt{count},r) & e=\mathtt{avg}(e')\\ w\text{ s.t. }\sigma\cup t,\delta\vdash e\Downarrow w\text{ and }t\in r & \text{o.w.}\end{cases}$$

REFERENCES

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 169–180. http://www.vldb.org/conf/2001/P169.pdf

Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen J. Westfold. 1998. Planware - Domain-Specific Synthesis of High-Performance Schedulers. In *The Thirteenth IEEE Conference on Automated Software Engineering, ASE 1998, Honolulu, Hawaii, USA, October 13-16, 1998*. IEEE Computer Society, 270. https://doi.org/10.1109/ASE.1998.732672

Peter A. Boncz and Martin L. Kersten. 1999. MIL Primitives for Querying a Fragmented World. *VLDB J.* 8, 2 (1999), 101–119. https://doi.org/10.1007/s007780050076

Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. In *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings (Lecture Notes in Computer Science)*, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh (Eds.), Vol. 4619. Springer, 139–150. https://doi.org/10.1007/978-3-540-73951-7_13

Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 227–238. https://doi.org/10.1145/1066157.1066184

Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, Alberto O. Mendelzon and Jan Paredaens (Eds.). ACM Press, 34–43. https://doi.org/10.1145/275487.275492

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 3–14. https://doi.org/10.1145/2491956.2462180

Rada Chirkova and Michael R. Genesereth. 2000. Linearly Bounded Reformulations of Conjunctive Databases. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings (Lecture Notes in Computer Science)*, John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.), Vol. 1861. Springer, 987–1001. https://doi.org/10.1007/3-540-44957-4_66

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

E. F. Codd. 1971. A Database Sublanguage Founded on the Relational Calculus. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, USA, November 11-12, 1971*, E. F. Codd and A. L. Dean (Eds.). ACM, 35–68.

Transaction Processing Performance Council. 2008. TPC-H Benchmark Specification. 21 (2008), 592–603.

Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2009. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_97.pdf

Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. 2011. *CMPH: C Minimal Perfect Hashing Library*. http://cmph.sourceforge.net

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 689–700. https://doi.org/10.1145/2676726.2677006

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1997. Index Selection for OLAP. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and Per-Åke Larson (Eds.). IEEE Computer Society, 208–219. https://doi.org/10.1109/ICDE.1997.581755

Angélica García Gutiérrez and Peter Baumann. 2007. Modeling Fundamental Geo-Raster Operations with Array Algebra. In *Workshops Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007), October 28-31, 2007, Omaha, Nebraska, USA*. 607–612. https://doi.org/10.1109/ICDMW.2007.53

Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294. https://doi.org/10.1007/s007780100054

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2010. Data Structure Fusion. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.), Vol. 6461. Springer, 204–221. https://doi.org/10.1007/978-3-642-

17164-2_15

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2011. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 38–49. https://doi.org/10.1145/1993498.1993504

Matthias Jarke and Jürgen Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16, 2 (1984), 111–152. https://doi.org/10.1145/356924.356928

Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *Proc. VLDB Endow.* 7, 10 (2014), 853–864. https://doi.org/10.14778/2732951.2732959

Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 958–968. https://doi.org/10.1145/3180155.3180211

Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 355–368. https://doi.org/10.1145/2908080.2908122

Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings (LNI)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.), Vol. P-241. GI, 383–402. https://dl.gi.de/20.500.12116/2418

Rachel Pottinger and Alon Y. Levy. 2000. A Scalable Algorithm for Answering Queries Using Views. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang (Eds.). Morgan Kaufmann, 484–495. http://www.vldb.org/conf/2000/P484.pdf

Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. (2013), 519–530. https://doi.org/10.1145/2491956.2462176

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 2–9. https://doi.org/10.1145/2784731.2784760

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

Michael Stonebraker. 1974. The choice of partial inversions and combined indices. *International Journal of Parallel Programming* 3, 2 (1974), 167–188. https://doi.org/10.1007/BF00976642

Michael Stonebraker. 2012. SciDB: An Open-Source DBMS for Scientific Data. *ERCIM News* 2012, 89 (2012). http://ercim-news.ercim.eu/en89/special/scidb-an-open-source-dbms-for-scientific-data

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 553–564. http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf

Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 1–25. https://doi.org/10.1145/2584665

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 307–322. https://doi.org/10.1145/3183713.3196893

Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias F. Stallmann. 2008. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings (ACM International Conference Proceeding Series)*, Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu (Eds.), Vol. 261. ACM, 311–322. https://doi.org/10.1145/1353343.1353383

Eelco Visser. 2005. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.* 40, 1 (2005), 831–873. https://doi.org/10.1016/j.jsc.2004.12.011

Cong Yan and Alvin Cheung. 2019. Generating Application-specific Data Layouts for In-memory Databases. *Proc. VLDB Endow.* 12, 11 (2019), 1513–1525. https://doi.org/10.14778/3342263.3342630

Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API discovery from demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 64–78. https://doi.org/10.1145/3062341.3062386