DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD

DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Toward a verified relational database management system

Citation

Malecha, Gregory, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. "Toward a verified relational database management system." In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 10, 237. Association for Computing Machinery. doi:10.1145/1706299.1706329. http://dx.doi.org/10.1145/1706299.1706329.

Published Version

doi:10.1145/1706299.1706329

Permanent link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:11318529

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Toward a Verified Relational Database Management System *

Gregory Malecha Greg Morrisett Avraham Shinnar Ryan Wisnesky
Harvard University, Cambridge, MA, USA
{gmalecha, greg, shinnar, ryan}@cs.harvard.edu

Abstract

We report on our experience implementing a lightweight, fully verified relational database management system (RDBMS). The functional specification of RDBMS behavior, RDBMS implementation, and proof that the implementation meets the specification are all written and verified in Coq. Our contributions include: (1) a complete specification of the relational algebra in Coq; (2) an efficient realization of that model (B+ trees) implemented with the Ynot extension to Coq; and (3) a set of simple query optimizations proven to respect both semantics and run-time cost. In addition to describing the design and implementation of these artifacts, we highlight the challenges we encountered formalizing them, including the choice of representation for finite relations of typed tuples and the challenges of reasoning about data structures with complex sharing. Our experience shows that though many challenges remain, building fully-verified systems software in Coq is within reach.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods, reliability; H.2.4 [*Database Management*]: Relational databases, query processing

General Terms Languages, Verification

Keywords relational model, dependent types, separation logic, B+ tree

1. Motivation

Relational database management systems (RDBMSs) have become ubiquitous components of modern application software. For example, SQLite, a lightweight RDBMS, ships as a component of Firefox, Skype, SymbianOS, and McAfee Antivirus, among others. In many of these applications, the RDBMS is used to store data whose *integrity* and *confidentiality* must be strictly maintained (e.g., financial records or security credentials). In an ideal world, an application developer would be provided with a high-level specification for the behavior of the data manager, suitable for formal (i.e., mechanical) reasoning about application-level security and correctness properties. Furthermore, the implementation of the data manager would be proven correct with respect to this specification to ensure that a bug cannot lead to accidental corruption or disclosure. It is for these reasons that we see *verified* RDBMSs as a compelling challenge to the programming languages and software verification communities that moves beyond the now successful domains of verified compilers and theorem provers.

As a step towards this goal, we have constructed a verified, lightweight, in-memory RDBMS using the Coq proof assistant [2]. Currently, our RDBMS supports queries, written in a stylized subset of SQL, over an in-memory relational store that can be [de]serialized to disk. As such, it provides much of the functionality needed for single-threaded client applications, but lacks the "ACID" properties (Atomicity, Consistency, Isolation, Durability) necessary in a concurrent, persistent storage system. The relational store is modeled using finite sets of typed tuples, and query semantics are expressed in terms of this model. Before execution, a query is transformed by a simple and provably semantics-preserving optimizer. The resulting optimized query is then mapped to a sequence of low-level operations over B+ trees. We implement B+ trees using Ynot [15], an axiomatic extension to Coq that provides facilities for writing and reasoning about imperative, pointer-based code.

The design and implementation are highly modularized to support code (and proof) re-use, and to enable alternative implementations. For example, the query execution engine works in terms of a generic finite map interface that can be realized with hash tables or other data structures besides B+ trees. As another example, the query optimizer can be extended with semantic optimizations that exploit *a priori* knowledge about relations, as long as appropriate (semantic) certificates of that knowledge can be presented to it.

The goal of this paper is to describe our verified, lightweight RDBMS and discuss the challenges we faced using a proof development environment such as Coq to build the system. Foremost among these were:

- Choosing an appropriate encoding of the relational model. Informally, a relation is simply a finite set of tuples over some basic value types. In Coq, there are many ways to represent such relations, each with different tradeoffs. For example, previous work encoding relational algebra in Agda suggests that schemas should be represented as functions from a finite set of column names to basic types [9], but in practice, we found that a concrete encoding using a list of type names yields a more workable representation.
- Another choice was how to represent finite sets. Finite sets are a common abstraction and Coq conveniently provides them as a standard library. Unfortunately, the library is currently coded as a compile-time, ML-style functor parameterized by a fixed element type. This is too restrictive for our RDBMS, which must determine the type parameter at run-time.
- Finally, we found formalizing correctness proofs for complicated, pointer-based data structures particularly difficult despite previous work in this area [3, 23]. B+ trees, which are gener-

* This research was supported in part by National Science Foundation grant 0910660 (Combining Foundational and Lightweight Formal Methods to Build Certifiably Dependable Software), NSF grant 0702345 (Integrating Types and Verification), and a NSF Graduate Research Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'10, January 17–23, 2009, Madrid, Spain.
Copyright © 2009 ACM 978-1-60558-479-9/10/01. . . $10.00

alized binary search trees with leaves connected by a linked-list "skirt" for rapid traversal, are often used to index data and are crucial for query execution performance. However, they are tricky to get right. BerkeleyDB, a high-performance embedded RDBMS that has hosted Google's accounts information [20], experiences around two dozen B+ tree-related bugs per version, according to its change-log.

The Ynot extension to Coq was designed to support writing and proving correct this kind of pointer-based code, using a variant of separation logic [18, 22]. Separation logic makes it particularly easy to reason about data structures with "local" pointer structure, such as trees. However, in the case of B+ trees, the pointer structure does not directly fit this pattern, which leads to complicated invariants and proofs of correctness.

Industrial strength RDBMSs include features which we have not yet implemented, such as indices and sophisticated query planning, as well as features which we can not yet reason about using our system. Nevertheless, the above challenges must be addressed before more sophisticated implementations can be verified.

Outline

This paper is structured as follows. In the next section we give an overview of what our RDBMS does, what we verify, and how the RDBMS is implemented. The sections after that present the specification and implementation in detail. We then discuss what development was actually like, give the lessons we learned, and provide measurements about verification overhead. We conclude with comparisons to related formalizations of the relational algebra in Agda. The source code is available at `http://ynot.cs.harvard.edu`. Note that for purposes of exposition we will sometimes omit inferable arguments and take other notational liberties.

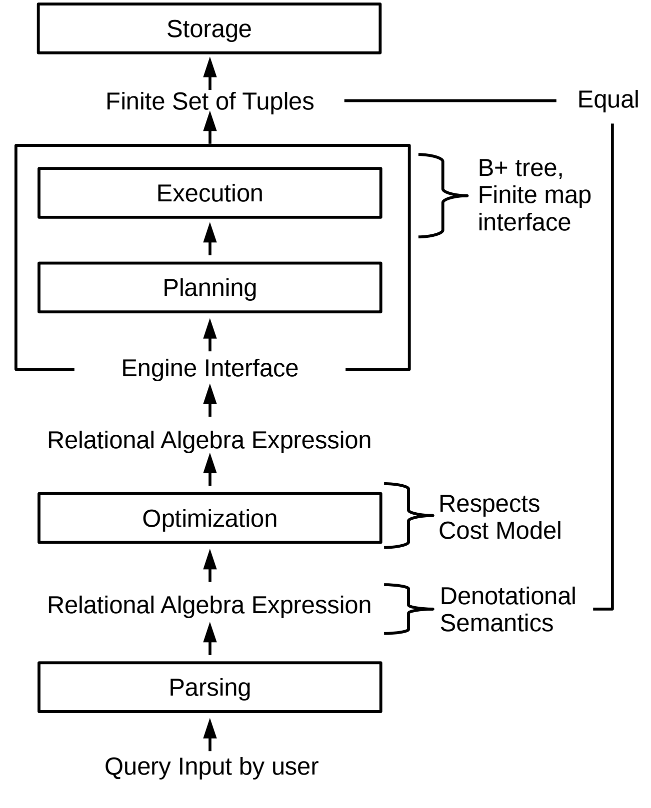

Figure 1. RDBMS Architecture.

2. Overview

We have constructed a simple, fully verified, in-memory RDBMS. A command line interface lets users create tables, load tuples into a table, save/restore a table to/from disk, and query the tables using a subset of SQL. The main verification task is showing that the RDBMS correctly executes queries with respect to a denotational semantics of SQL and relations. Execution includes parsing SQL concrete syntax into abstract syntax trees (ASTs), transforming the ASTs into relational algebra expressions, performing source-to-source optimizations on the relational algebra, and then interpreting queries as series of operations over B+ trees, as shown in Figure 1. The RDBMS has five main components, each of which has been implemented in Coq:

- The relational algebra model (Section 3) defines schemas, relations, and declarative specifications of query operations.
- The SQL compiler (Section 4) includes the parser, definitions for SQL abstract syntax, a denotational specification for SQL in terms of the model, and semantics-preserving SQL optimizations. We also formulated a run-time cost model and proved that several transformations are not cost-increasing.
- The SQL execution engine (Section 5) interprets the optimized SQL expression as a series of operations over imperative finite maps. Correctness is established using Hoare-style reasoning relating imperative finite maps to the relations they represent.
- The B+ tree implementation (Section 6) provides finite map operations for insertion and lookup of key-value pairs, and iteration (amongst others). These imperative operations are coded with the Ynot extension to Coq and verified using a variant of separation logic.
- The storage interface (Section 7) is responsible for [de]serializing relations to disk and establishing integrity constraints. The storage manager includes a proof that deserializing the serialized form of a relation R results in R, under the assumption that disk operations do not fail.

The Coq extraction facility allows us to run our code by translating it to OCaml. During this process, non-computational content used only in specifications, such as the relational model, is erased. The Ynot library provides imperative OCaml definitions for Coq axioms (e.g., Ynot references become OCaml references manipulated with standard effectful OCaml read/write operations).

3. The Model

The relational algebra has a standard definition in terms of set theory, so a large portion of our RDBMS specification deals with realizing both sets and the relational algebra in Coq. We begin with an informal overview of relational concepts and then discuss how each of these is realized within Coq.

As is standard, we model a database using *relations*. A relation can be represented as a finite set of tuples over a list of primitive types. It is helpful to think of a relation as a table (with no duplicate rows) where the rows represent entries and the columns represent attributes of an entry. The list of primitive types that describes the columns is known as the *schema* for the relation.

Tuples in a relation are indexed by a set of *attribute names*. To simplify our implementation, we use the position (column) of an element as the attribute name, but it would be easy to support another index set by maintaining a mapping from names to offsets.

New relations are constructed using a basis of operations:

- **Selection**. Given a predicate P, selection returns the subset of the relation's tuples that satisfy P.
- **Projection**. Restricts each tuple in a relation to a subset of the relation's columns.
- **Permutation**. Permutes a relation's columns.
- **Union**. Returns the set-theoretic union of two relations.
- **Difference**. The difference of relation A and B returns the result of removing every tuple in B from A.
- **Cartesian Product**. The cartesian product of relation A with n columns and relation B with m columns consists of every $(n + m)$-tuple that can be formed by concatenating a tuple in A with a tuple in B.

This basis is *relationally complete*, and equal in expressive power to other relational formalisms, like relational calculus [1]. However, the choice of basis is somewhat arbitrary; for example, we could have chosen to use join and recovered cartesian product as a degenerate case.

The ordered, nameless schema representation is a design decision that we believe simplified reasoning and implementation. To accommodate an unordered, named representation would require a renaming operation instead of permutation.

3.1 Schema

In our model, the schema for a relation is defined as follows:[1]

```
Parameter tname : Set.
Parameter tnameDenote : tname -> Set.

Definition Schema : Set := list tname.
```

Thus, a schema is represented as a list of type names, and type names can be mapped to Coq types by a denotation function. The definition for type names and the denotation function are parameters to the system so that users can easily add new constructors to the set of schema types. For example, we might define `tname` as the inductive definition:

```
Inductive tname : Set :=
| Nat : tname | Bool : tname | Str : tname
| Option : tname -> tname.
```

and define the denotation function as:

```
Fixpoint tnameDenote (t:tname) : Set :=
  match t with
    | Nat => nat
    | Bool => bool
    | Str => string
    | Option t' => option (tnameDenote t')
  end.
```

where `nat`, `bool`, etc. are the corresponding Coq types. The values that make up tuples are inhabitants of the denoted Coq types. For example,

```
"Abc" : tnameDenote Str  and  17 : tnameDenote Nat
```

We need to be able to compare schemas for equality (e.g., to check that an optimization is admissible). Equality between arbitrary Coq types is undecidable, so we require decidable equality on type names as another parameter to the system:

```
Parameter tname_dec_eq :
  forall (n1 n2: tname), {n1=n2} + {n1<>n2}.
```

[1] Here `Set` refers to Coq's type universe, not sets in the sense of relations.

Additionally, we require that for any type name, the Coq type T that it denotes satisfies the following properties:

1. T must be a decidable setoid; that is, come equipped with a decidable equivalence relation.
2. T must be a decidable total order; that is, come equipped with a decidable total ordering compatible with the setoid.
3. T must be serializable; that is, come equipped with a pair of functions `ser` : $T \rightarrow$ `string` and `deser` : `string` $\rightarrow$ `option` T such that $\forall x{:}T$, `deser`(`ser` x) = `Some` x.

Like decidable equality for type names, these properties on denotations are given as parameters to the system. Property (1) allows for equivalence relations on column types that are weaker than syntactic (Leibniz) equality. For instance, we can treat strings in a case-insensitive way. Property (2) is required because of the way we build sets of tuples, and property (3) is used for persistence.

3.2 Tuples

As we have seen, each column in a relation has a type, and a tuple associates a value of the appropriate type to each column. That is, a tuple is a heterogeneously-typed list. The type of a tuple is given by a recursive, type-level function (where `::` is list cons) :

```
Fixpoint Tuple (A: Schema) : Set :=
  match A with
   | nil => unit
   | n :: t => tnameDenote n * Tuple t
  end.
```

Tuples are essentially iterated pairs of values, terminated by a unit (the single inhabitant of `unit` is written `tt`). For example,

```
Definition aSchema : Schema :=
  Str :: Nat :: Bool :: nil.

Definition aTuple : Tuple aSchema :=
  ("Hello world", (17, (true, tt))).
```

To express the relational operations we need to define several tuple manipulation functions. For example, to perform product we need an operation with the following type to fuse tuples:

```
fuseTuples (I J: Schema)(x: Tuple I)(y: Tuple J) :
  Tuple (I ++ J).
```

The type of this function ensures that the schema of the resulting fused tuple is the concatenation of the input schemas. We also use the richness of Coq's type system to help simplify reasoning about error cases. For instance, to project out the type name of a particular column n from a schema I, we need to provide a proof that witnesses that n is less than the length of I:

```
colType (I:Schema) (n:nat) (pf:n<length I) : tname.
```

The operation to project a single column from a tuple uses `colType` in its type:

```
projTupleCol (I: Schema) (n: nat) (pf: n < length I)
  (t: Tuple I) : tnameDenote (colType I n pf).
```

We implement multi-column projection by iterating `colType` and `projTupleCol` to obtain `colTypes` and `projTupleCols`.

3.3 Relations as Finite Sets

The next choice we consider is how to represent finite sets in Coq. Coq provides the "FSets" library for this, but we could not use it directly. The library is coded as an ML-style functor which requires the static determination of the carrier. Our RDBMS must compute this type from a schema at run-time, because it does not know table

schemas until the user actually loads data at run-time. Rather than try to encode such behavior using modules, we modified the FSet library to be first-class using Coq's type class mechanism.

Type classes are a recent addition to Coq and behave similarly to their Haskell counterparts [27]. They allow the user to overload a set of operations across a class of types. We have a class of types `FSetInterface` that is parameterized by a type `elt` of elements and a total ordering `E` over `elt` that can be used as specifications of finite sets. Here `Prop` indicates Coq's type of computationally irrelevant propositions:

```
Class FSetInterface (elt: Set) (E: OrderedType elt)
  : Type :=
{ Fset: Set; (* the abstract type of finite sets *)

  (* operations *)
  empty : Fset;
  union : Fset -> Fset -> Fset;
  is_empty : Fset -> bool;
  ...
  (* predicates *)
  In : elt -> Fset -> Prop;
  Equal := fun s s' => forall a : elt,
    In a s <-> In a s';
  ...
  (* axioms *)
  union_1 : forall s s' x, In x (union s s') ->
    In x s \/ In x s';
  union_2 : forall s s' x, In x s ->
    In x (union s s');
  ...
}.
```

In addition to specifying the operations that define the class, we specify a set of axioms that allow us to reason about the operations. We must also show that this specification is realizable, which we do by providing a simple implementation using lists.

Given that we require the element types to be ordered, an alternative implementation of a finite set would be as a sorted list with a proof that the list contains no duplicates. The advantage of this concrete representation, over our axiomatic interface, is that we could: (1) derive the axioms directly from the representation; and (2) rewrite directly using Leibniz equality. The disadvantage, of course, is that the interface would not be as re-usable in other application contexts where we desire a weaker notion of equality.

In general, we have found that the richness of Coq, including support for ML-style modules, dependent records, and type classes, coupled with abstraction and equality issues yields a set of design tradeoffs that are difficult to evaluate without exploring (by coding) many alternatives. One set of tradeoffs is present in any RDBMS formalization; for instance, whether to name columns or use column numbers. Another set of tradeoffs arises only because certain forms of reasoning appear to be more effective in Coq and would not appear in a paper and pencil formalization; for instance, the choice of axiomatic sets instead of sets as concrete lists. Both sets of tradeoffs are equally important for verified software because every detail must be checked. Unfortunately, it can often take significant time to fully understand the Coq-engineering consequences of a seemingly inconsequential design choice; for instance, whether to represent schema as lists of type names or as functions from column numbers to type names. At present it is unclear how best to explore the design space.

3.4 Relational Algebra in Coq

We define the relational operations over finite sets of schema-typed tuples (relations). Building relations requires defining a total ordering over tuples and interacting with the type class mechanism, but, for purposes of exposition, we essentially have that:

```
Definition Relation (I:Schema) :=
  FSetInterface (Tuple I).
```

Union, difference, and selection are implemented in terms of the `FSetInterface` `union`, `difference`, and `filter` functions. Projection and product are defined using the generic `fold` function provided by the `FSetInterface`.

Selection allows any Coq predicate that respects the setoid equality of the schema column types to be used, but our SQL syntax is less expressive. Projection is implemented by iterating through a set, projecting out each tuple individually. Cartesian product is only slightly more complicated, requiring two iterations. To compute the product of A and B, for each $a \in A$ we compute the set $\{$ `fuseTuples` $a\ b \mid b \in B \}$ and then union the results.

In addition to defining the relational operations, we need a number of lemmas to support basic reasoning. For instance, when we project out column n from schema I, using a proof pf that n is less than the length of I, we must demonstrate that we get the same result as when we use a different proof pf'. (This lemma is proof irrelevance for `colType`). These foundational lemmas are typically easy to prove but nonetheless required. Similarly, our use of dependent types means that we often run into situations where we would like schemas to be definitionally equal but they are only propositionally equal, and so we are forced to reason about equality explicitly. This leads to more verbose theorem statements and proofs.

To gain confidence in the accuracy of our model of the relational algebra, we have shown that several dozen standard equivalences are derivable from our definitions. We also use these identities to justify the correctness of our query optimizations.

Some equivalences are universally valid; for example, the commutativity of selection:

```
select P1 (select P2 R) = select P2 (select P1 R)
```

Other equivalences only apply in the presence of constraints on the input relations. For instance, let A and B be relations over schemas I and J, respectively, and let l indicate the columns $0...|I|-1$. Then we have the following conditional equivalences:

```
B <> empty -> proj l (prod A B) = A
B  = empty -> proj l (prod A B) = empty
```

Proving this statement requires reasoning about how projection of l can be pushed through the nested iteration that defines the product. The actual proof proceeds much like on paper: first we demonstrate that every element in the product of A and B must be a fused tuple of the form `fuseTuples` $x\ y$, and that fusing x with y followed by projection by l yields x. The theorem then follows from showing that every element of A has at least one corresponding fused tuple in the product of A and B, provided B is non-empty. We constructed proofs manually, but it may be possible to adapt an automated theorem prover for relations (e.g., [25]) to Coq to reduce the proof burden.

4. The SQL Compiler

In the previous section we described our Coq encoding of a *model* for relational data and algebraic operations on relations. In this section we describe the front-end of our SQL implementation and relate it to this model.

4.1 Abstract Syntax

We define a subset of SQL abstract syntax with a Coq data type that is indexed by the schema of the relation that the expression denotes.

Variables, which are really table names, are represented as strings and are explicitly typed. The overall effect is that queries are well-typed by construction and coupled with our other design decisions means that they can always be given a meaning in terms of the relational algebra. Queries are defined by the following inductive data type:

```
Inductive Query : Schema -> Set :=
| varExp : forall I, string -> Query I
| unionExp : forall I,
    Query I -> Query I -> Query I
| diffExp : forall I,
    Query I -> Query I -> Query I
| selectExp : forall I,
    whereExp I -> Query I -> Query I
| projExp : forall I (l:list nat)(pf: bounded I l),
    Query I -> Query (colTypes l pf)
| prodExp : forall (I J: Schema),
    Query I -> Query J -> Query (I++J).
```

The projection expression requires a proof that the columns l that define the projection are each less than the length of I. Selection is defined using an additional syntax for predicates over tuples. Our definition of selection lets us use arbitrary Coq functions (that respect the setoid equality), but we restrict users to only certain kinds of predicates (whereExps). We support boolean combinations of comparisons between columns and constants. That is,

```
Inductive atomicExp (I : Schema) : tname -> Set :=
| const: forall t (c:tnameDenote t), atomicExp I t
| col  : forall n (pf: n < length I),
    atomicExp I (colType I n pf).

Inductive compareExp I : Set :=
| compareEq : forall t, atomicExp I t ->
    atomicExp I t -> compareExp I
| compareLt ...

Inductive whereExp I :=
| compExp: compareExp I -> whereExp I
| andExp : whereExp I -> whereExp I -> whereExp I
| orExp  ...
```

4.2 Parsing

The RDBMS uses Ynot's packrat PEG parser [5] to parse user input. The parser is implemented as a verified compiler [12]: given a specification consisting of a PEG grammar and semantic actions, the parser creates an imperative computation that, when run over an arbitrary imperative character stream, returns a result that agrees with the specification. To make the parsing efficient, the packrat algorithm used by the resulting computation uses a sophisticated caching strategy that is implemented using imperative hash tables.

Queries written in our abstract syntax resemble SQL statements, but to support some features of SQL our system must express the operations in terms of the basis we have chosen. SQL supports, for instance, syntax for both cartesian product and join, but our basis only has product. Hence an SQL-ish join query, such as:

```
SELECT 0, 1, 2 FROM
   (JOIN tbl1 , tbl2 ON col 0 = col 0)
WHERE col 0 = "hello world" AND col 1 < col 3
```

is translated by our system into a product, selection, and projection:

```
SELECT 0, 1, 2 FROM
   (SELECT 0, 1, 2, 4, 5
    FROM tbl1, tbl2 WHERE col 0 = col 3)
WHERE col 0 = "hello world" AND col 1 < col 3
```

assuming that tbl1 has 3 columns (indexed 0-2 in the product). During query optimization, selection fusion will optimize this query (Section 4.4).

4.3 SQL Semantics

To give a query a meaning in terms of the relational algebra we need to know the relations that correspond to the query's variables. These relations correspond to the actual data that the user has loaded, and the association between the variables and the tables is captured using the traditional mechanisms of a context and an environment:

```
Definition Ctx := list (string * Schema).

Fixpoint Env (G: Ctx) : Set :=
  match G with
    | nil => unit
    | (_, J) :: b => Relation J * (Env b)
  end.
```

When an environment has a (properly-typed) relation corresponding to each variable in a query (which is easy to check), the denotational semantics of a query is defined by recursively applying the relational operations or looking up the value of a table in the environment as necessary:

```
Fixpoint denote (I: Schema) (q: Query I)
 (G: Ctx) (E: Env G) : Relation I :=
  match q with
    | varExp J v => lookup E I v
    | unionExp J a b =>
       union (denote a) (denote b)
    | diffExp J a b =>
       diff  (denote a) (denote b)
    | selectExp J r f =>
       select (whereDenote f) (denote r)
    | projExp J l pf e => proj l pf (denote e)
    | prodExp I' J' a b =>
       prod  (denote a) (denote b)
  end.
```

This definition uses an auxiliary denotation function whereDenote: whereExp I -> Tuple I -> bool with the obvious definition.

4.4 Source-to-Source Optimization

A query optimization is a semantics-preserving source-to-source transformation (over the typed SQL syntax) that ideally reduces execution time. For instance, common optimizations include reducing the number of joins in a query and pushing selection towards the leaves. These work well in practice because joins tend to dominate query execution time, and selection tends to reduce relation size.

In our system optimization is a distinct phase before execution. We have implemented a number of optimizations, including the following three from a standard introductory database textbook [6]:

- **Selection fusion.** The query `select P2 (select P1 Q)` can be transformed to a single selection: `select (P1 and P2) Q`.
- **Projection fusion.** The query `proj l2 (proj l1 Q)` can be transformed to use a single projection: `proj (l2 ∘ l1) Q`, where `(l2 ∘ l1)` is the composition of permutations `l1` and `l2`.
- **Selection-Projection re-ordering.** The query `select P (proj l Q)` can be transformed to `proj l (select P' Q)`, where `P'` is `P` extended to operate over tuples with additional columns.

In principle any of the several dozen algebraic identities that we have proven can be used as the basis of a transformation, although not all will be beneficial.

4.4.1 Cost Estimation

We would like a lightweight way to check whether or not our optimizations are likely to reduce running time. We can estimate the execution time and resulting relation size for a query by making conservative assumptions about the data and the algorithms being used to implement the relational operations. Currently, we assume that selection and projection must iterate through every tuple in a relation, and that computing the union, difference, or product of two relations requires comparing every tuple in both relations. Of course, these assumptions are overly pessimistic – in practice, union can be an $O(n+m)$ operation assuming the data are sorted, not an $O(n \cdot m)$ one – but we are only interested in the *difference* between the running time of the query before and after optimization. Because this simple cost model is data-independent, it does not incorporate useful assumptions such as how selection decreases the cardinality of relations. However, even using this simple model we can still show that some of our optimizations, like selection fusion, do not increase cost.

4.4.2 Semantic Optimization

Semantic optimizations [8] are rewrites that are semantics-preserving only because of the specific constraints on the particular database instance at hand. Our "proj-prod" equivalence for relations A and B of schemas I and J where l is the columns $0...|I|-1$:

```
B <> empty -> proj l (prod A B) = A
B  = empty -> proj l (prod A B) = empty
```

can be used as such an optimization because which rewrite to perform depends on whether or not B is empty. Our RDBMS tracks the emptiness of user tables through load operations and provides this information to a semantic optimization, which can rewrite and witness the semantics preservation with the available emptiness proof. Semantic information about relations is maintained in `RelInfo` records:

```
(* Relation information *)
Record RelInfo := relInfo {
  isEmpty : bool
}.

(* Meta data for each table *)
Definition DbInfo : string -> Schema -> RelInfo.
Definition ra_sem_rewrite : Set :=
 DbInfo -> (forall I, Query I -> Query I).
```

In the future, this record can be elaborated with additional statistics, such as the number of tuples in the relation and selectivity properties of certain columns, allowing the optimizer to optimize queries more intelligently. For example, we might choose an order for symmetric operations so as to iterate over the smaller relation.

The optimization associated with the "proj-prod" equivalence identifies expressions of the form `proj` l (`prod` q v), where q is a query over schema I, and v is a variable of schema J. The optimization uses the `DbInfo` to decide which rewrite to apply.

We need one final component to ensure that semantic optimizations are used correctly: we need to make sure that the `RelInfo` information associated with the database is accurate. This is necessary to prove that the semantic optimizations are in fact meaning preserving. Therefore we guarantee that our database is always in a state such the following invariant holds:

```
Definition accurate (m: DbInfo)
  (G: Ctx) (E: Env G) : Prop :=
  forall s I, getRelation s I E = empty <->
              isEmpty (m s I)
```

Semantic optimizations are well studied but have traditionally been difficult to deploy. In [8] the authors argue that the limiting factor for industrial deployment of semantic optimizations is the lack of support for explicitly manipulating symbolic constraints. One benefit of our formalization is that semantic optimization becomes less risky because we are required to prove semantic preservation formally.

5. The SQL Execution Engine

Once a query has been optimized, our RDBMS executes the query using a sequence of operations over imperative *finite maps*. Anticipating the use of keys for indexed retrieval (see Section 5.3), we use maps from keys to values, instead of sets. Our finite maps are implemented using B+ trees as described in Section 6, but the engine is insulated from this detail by working in terms of a generic interface. Indeed, it is possible to replace the implementation with an alternative, such as a finite map based on hash tables. In this section, we describe the finite map interface, how SQL queries can be naïvely implemented in terms of the interface, and how key constraints can be utilized to optimize query execution.

5.1 The Finite Map Interface

The finite map interface is specified with two components: the first is a *functional* specification that describes finite maps as simple association lists that map keys to values. The second is an *imperative* specification, in the style of Hoare Type Theory (HTT) [14] and realized by the Ynot extension to Coq [15]. In an HTT specification, we describe operations over an imperative Abstract Data Type (ADT) using commands indexed by pre- and post-conditions. We use the functional model to describe the state of the ADT before and after the command is executed and to relate the result of the command to the pre- and post-state of the ADT.

The functional state of a finite map is given by a sorted association list of key-value pairs. The ADT operations, such as `lookup` and `insert`, have simple definitions as pure Coq code:

```
Parameters key value : Set.
Definition AssocList := list (key * value).

Fixpoint specLookup (k : key) (m : AssocList)
  : option value :=
  match m with
    | nil => None
    | (k',v) :: b => if k' = k then Some v
                     else specLookup k b
  end.

Fixpoint specInsert (k : key) (v : value)
  (m : AssocList) : AssocList :=
  match m with
    | nil => (k, v) :: nil
    | (k',v') :: b =>
       match compare k' k with
         | LT => (k',v') :: specInsert k v b
         | EQ => (k ,v ) :: b
         | GT => (k ,v ) :: (k',v') :: b
       end
  end.
```

The imperative interface is shown in Figure 2. It specifies an abstract type `handle` that represents a handle on the imperative state of the finite map. Next, the interface requires a predicate `rep` that connects a handle to an association list in a particular state. The intention is that the predicate `rep` h ℓ should hold in a heap when the ADT h *represents* the association list ℓ.

```
(* The abstract type of imperative finite maps. *)

Parameter handle : Set.

(* The predicate rep h m holds in a heap when the finite map h represents the association list m. *)

Parameter rep : handle -> AssocList key value -> heap -> Prop.

(* Newly created finite maps are empty (emp is the empty heap assertion in separation logic), and so represent the nil association list. *)

Parameter new : Cmd emp (fun h : handle => rep h nil).

(* Destroy a finite map, reclaiming the memory used. The return type is unit. *)

Parameter free : forall (h : handle) (m : AssocList), Cmd (rep h m) (fun _:unit => emp).

(* The function lookup h k can be run in a state where h represents some model m, and it returns the value associated with k in m. In the
post-state, h continues to represent m.*)

Parameter lookup : forall (h : handle) (k : key) (m : AssocList),
  Cmd (rep h m) (fun res : option value => rep h m * [res = specLookup k m]).

(* The function insert h k v can be run in a state where h represents some model m. In the post-state, h represents m extended to map k to
v. The operation returns the old value associated with k (if any). *)

Parameter insert : forall (h : handle) (k : key) (v : value) (m : AssocList),
  Cmd (rep h m) (fun res : option value => rep h (specInsert v m) * [res = specLookup k m]).

(* Fold a command over the elements of the map. The I serves as a loop invariant on the command, relative to the list of elements that have
been touched at each step in the computation. *)

Parameter iterate : forall (T : Type) (h : handle) (I : T -> AssocList -> heap -> Prop)
  (tm : AssocList) (acc : T)
  (fn : forall (k: key) (v: value) (acc: T) lm, Cmd (I acc lm) (fun a:T => I a (lm ++ (k, v) :: nil))),
  Cmd (rep h tm * I acc nil) (fun res:T => rep h tm * Exists tm', I res tm' * [Permutation tm tm']).
```

Figure 2. The Imperative Finite Map Interface.

The rest of the interface gives the specifications for the finite map operations. The specifications are encoded using a type `Cmd` of *imperative commands* (also called *computations*) that are indexed by pre- and post-conditions over the heap. `Cmd` is similar to the `IO` and `ST` monads in Haskell except that the indices capture the behavior of the command in a Hoare-logic style. The intention is that given a `Cmd` $P\,Q$, we can run the command in any state satisfying P, and if that command terminates, then we are assured that Q holds on the final state. Furthermore, we are guaranteed that the computation will not terminate prematurely due to type errors or other run-time errors such as null-pointer dereferences.

The pre- and post-conditions are specified using a variant of *separation logic* [18, 22], and commands are constructed to satisfy the frame rule with respect to separating conjunction. In other words, if c : `Cmd` $P\,Q$, then c also has type `Cmd` $(P * R)\,(Q * R)$ for any heap predicate R, where "$*$" is separating conjunction.

For example, the `new` command specifies `emp` as a pre-condition and thus demands that the heap is empty. It returns a handle `h` and terminates in a state where h represents the empty association list. Because `new` satisfies the frame rule, it can be run in any heap satisfying a property R, and we are ensured that R continues to hold after executing the command.

The `free` rule is dual to `new`. It requires that we pass in a handle which, in the initial state, represents some association list m. The post-condition specifies that the heap will then be empty. But as before, this can be run in any larger, disjoint heap satisfying a predicate R, and we are ensured R continues to hold after execution.

The `lookup` operation takes a handle representing association list m and a key k and returns an optional value. The post-condition captures the facts that: (1) the returned value is equivalent to doing a functional lookup on the model list m; and (2) the handle continues to represent m in the post-state. The brackets around the expression "`res = specLookup k m`" are notation for lifting a pure predicate (type `Prop`) to a predicate over heaps (type `heap -> Prop`).

The `insert` operation takes a handle h, key k, and value v such that h represents association list m on input, and it ensures that h represents the result of inserting (k, v) into m on output. The command also returns the old value (if any) that was assigned to k in the input state; this makes it easy to undo the insertion.

The `iterate` operation is a higher-order command that iterates over the elements of the finite map, applying another command fn to each key-value pair and an accumulator to yield a final value. The computation is parameterized by an invariant I which is used to accumulate logical results about the iteration of fn. The separating conjunction ensures that fn cannot change the map during iteration.

In the actual code, the association list parameters are marked as computationally irrelevant using the approach described in our previous work on Ynot [5]. This ensures that these values are only used in types and specifications and do not affect the behavior of programs. Consequently, the extracted ML code does not need to include them, and these model values incur no run-time overhead.

5.2 Interpreting SQL Operations

Our SQL engine executes queries using the finite map interface. In an industrial RDBMS, the engine would be able to choose from multiple relation representations and algorithms (e.g. hash tables vs B+ trees, hash-join vs merge-join, etc.), and an input query would be explicitly lowered into another intermediate form that more closely corresponds to these implementation choices. This lowered form is also typically optimized again. In our case, however, we simply execute queries by directly interpreting them as sequences of finite map operations, as described below:

- **Union**. To union A and B, `iterate` through A, `inserting` each $(k, v) \in A$ into B.
- **Difference**. To subtract B from A, `iterate` through A, `inserting` each $a \in A$ into a new relation when $a \notin B$.
- **Projection**. To project columns from A, `iterate` through A, `inserting` the projected tuples into a new relation.
- **Selection**. To select rows from A, `iterate` through A, `inserting` tuples that meet the criteria into the new relation.
- **Product**. To compute the cartesian product of A and B, `iterate` through each $a \in A$ and, for each $a \in A$, `iterate` through each $b \in B$, `inserting` a fused tuple $a + b$ into the new relation.

5.3 Taking Advantage of Key Constraints

Key constraints are useful both for ensuring data integrity and for increasing performance. When a user loads a table into our RDBMS, they may indicate which of the columns are intended to form a *key* for the table. Given a relation R over schema I, a subset i of I's columns is a key for R iff for every two tuples $x, y \in R$, we have that `proj` $i\,x$ = `proj` $i\,y$ implies $x = y$. A good example of a key is a person's social security number; in principle, knowing this value is enough to uniquely identify a person. In general, a relation may have many keys, and the set of all of a relation's columns always forms a key. It is easy to check if a set of columns forms a key, and our RDBMS aborts the loading process if a key constraint is violated.

We are working toward (but have not completed) taking advantage of key constraints using the finite map interface to execute *point queries* in logarithmic time using `lookup`. A point query is a degenerate range selection [7] where the predicate uniquely identifies a tuple. The predicate must consist of a set of equalities between columns and values such that the columns form a key for the relation being scanned. For example, suppose we have a table with two columns, user name and password, where user name is a key. To determine a user's password, we might run the following query:

```
SELECT 1 FROM users WHERE col 0 = "Adam"
```

Since the name is a key we know that there exists at most one entry in the relation with the name "Adam". Furthermore, if the name is indexed, then we can directly call the finite map's `lookup` operation which answers the query in logarithmic time rather than linear time.

6. The B+ Tree Implementation

At the core of our RDBMS is an implementation of the finite map interface of Figure 2 using a B+ tree [1], a ubiquitous data structure when ordered key access is common. (In industrial RDBMSs, B+ trees are also used to build additional indices into relations). In this section we describe the B+ tree implementation as well as the choices we made for representing the structure in separation logic and reasoning about its correctness.

A B+ tree is a balanced, ordered, n-ary tree which stores data only at the leaves and maintains a pointer list in the fringe to make in-order iteration through the data efficient. Figure 3 shows a simple B+ tree with arity 3.

As with most tree structures, B+ trees are comprised of two types of nodes:

- Leaf nodes store data as a sequence of at most n key-value pairs in increasing order. Leaves use the trailing pointer position to store the pointer to the next leaf node.
- Branch nodes contain a sequence of at most n pairs of keys and subtrees. These pairs are ordered such that the keys in a subtree are less than or equal to the given key (represented in the figure as `treeSorted min max`). For example, the second subtree can only contain values greater than 3 and less than or equal to 5. In addition to the n sub-trees, branch nodes include a final subtree that covers the upper span. In the figure, this is the span greater than 5.

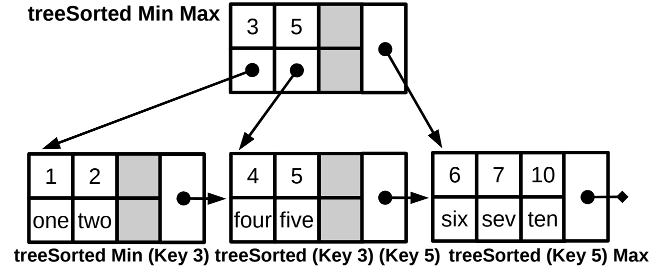

Figure 3. B+ tree of arity 3 with numeric keys and string values.

B+ trees must maintain several additional invariants needed to ensure logarithmic-time insertion and lookup. First, non-root nodes must contain at least $\lceil \frac{n}{2} \rceil$ keys. Second, if the root node is a branch, it must contain at least 2 children. Third, all subtrees of a branch must be the same height.

6.1 B+ Tree Representation and Invariants

In order to meet the finite map interface, we need to characterize B+ trees and their associated heaps. As previously mentioned, B+ trees are parameterized by an arity (previously n) which we will call `SIZE` which must be greater than 1, and, for simplicity when reasoning about division by 2, even:

```
Parameter SIZE      : nat.
Parameter SIZE_min  : 1 < SIZE.
Parameter SIZE_even : even SIZE.
```

In the heap, we store records that represent nodes in the tree. These are represented by the following Coq data type:

```
Record bpt_node : Set := mkBpt {
  height  : nat;
  content : array;
  next    : option ptr
}.
```

The `content` field stores an array of length `SIZE`; the type of values in the array are captured in the heap assertions. Following common C practice, we use the `next` field to encode the final child when the node is a branch and the next pointer in the linked-list of leaves when the node is a leaf.

Our next task is to define predicates that describe when a heap contains a tree with a valid shape. To simplify this task, we first define a functional model of the tree, which we call a `ptree`:

```
Fixpoint ptree (h : nat) : Set :=
  ptr * match h with
        | 0    => list (key * value)
        | S h' => list (key * ptree h') * ptree h'
       end.
```

In this definition, `ptree`s are indexed by a height `h`. When the height is 0, the `ptree` consists of a pointer and list of keys and associated values. When the height is $h' + 1$ for some h', the `ptree` consists of a pointer, a list of keys and `ptree`s of height h', and a final `ptree` of height h'. Our general strategy is to associate a `ptree` model with each B+ tree, to capture the salient aspects of its shape. We then impose well-formedness constraints on the model, such as the requirement that the keys in a node are sorted.

We say that a `ptree` p of height h is a model of a B+ tree with root pointer r when the predicate $\texttt{repTree}\ h\ r\ \texttt{None}\ p$ holds. This predicate is defined to capture the shape constraints of B+ trees as described above and uses two auxiliary predicates, `repLeaf` and `repBranch` discussed below:

$$\begin{array}{l}\texttt{repTree}\ 0\ r\ optr\ (p', ls) \iff \\ \quad [r = p'] * \exists ary.\ r \mapsto \texttt{mkNode}\ 0\ ary\ optr\ * \\ \qquad \texttt{repLeaf}\ ary\ |ls|\ ls\end{array}$$

$$\begin{array}{l}\texttt{repTree}\ (h+1)\ r\ optr\ (p', (ls, nxt)) \iff \\ \quad [r = p'] * \exists ary. r \mapsto \texttt{mkNode}\ (h+1)\ ary\ (\texttt{ptrFor}\ nxt) * \\ \qquad \texttt{repBranch}\ ary\ (\texttt{firstPtr}\ nxt)\ |ls|\ ls * \\ \qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ nxt)\ optr\ nxt\end{array}$$

The `repTree` predicate has two cases, depending upon the `ptree`'s height. In both cases, we require that the root of the B+ tree r is equal to the pointer recorded in the `ptree` (determined by `ptrFor`) and that r points to a `node` record with appropriate information, including an array. The `firstPtr` function computes the pointer of the first leaf in the given tree and is used to ensure the link between adjacent subtrees. When the `ptree` height is zero, we require that the array holds the list of key-value pairs in the `ptree` as defined by the `repLeaf` predicate:

$$\begin{array}{l}\texttt{repLeaf}\ ary\ n\ [v_1, ..., v_n] \iff \\ \quad ary[0] \mapsto \texttt{Some}\ v_1 * ... * ary[n-1] \mapsto \texttt{Some}\ v_n * \\ \quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}\end{array}$$

When the height of the `ptree` is non-zero, we require that the array holds a list of key-pointer pairs and that these are valid models for the key-sub-tree pairs specified by the `ptree`. These properties are captured by the `repBranch` predicate:

$$\begin{array}{l}\texttt{repBranch}\ ary\ n\ optr\ [(k_1, t_1), ..., (k_n, t_n)] \iff \\ \quad ary[0] \mapsto \texttt{Some}\ (k_1, \texttt{ptrFor}\ t_1) * \\ \qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_1)\ (\texttt{firstPtr}\ t_2)\ t_1\ * ... * \\ \quad ary[n-2] \mapsto \texttt{Some}\ (k_{n-1}, \texttt{ptrFor}\ t_{n-1}) * \\ \qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_{n-1})\ (\texttt{firstPtr}\ t_n)\ t_{n-1} * \\ \quad ary[n-1] \mapsto \texttt{Some}\ (k_n, \texttt{ptrFor}\ t_n) * \\ \qquad \texttt{repTree}\ h\ (\texttt{ptrFor}\ t_n)\ optr\ t_n * \\ \quad ary[n] \mapsto \texttt{None} * ... * ary[SIZE-1] \mapsto \texttt{None}\end{array}$$

These definitions are encoded as a set of mutually-recursive Coq fixpoints which describe, computationally, the heap-shape of a given `ptree`.

In order to iterate over the elements of the B+ tree, we only need access to the leaves. Consequently, in addition to the `repTree` predicate, we have defined predicates `repLeaves` and `repTrunk` which separate the tree into two disjoint components. This facilitates reasoning about operations such as iteration, which only need access to the leaves. A critical lemma connects the two views:

```
Theorem repTree_iff_repTrunk : forall h (r : ptr)
  (optr : option ptr) (p : ptree h) (H : heap),
  repTree r optr p H <->
  (repTrunk r optr p *
  repLeaves (Some (firstPtr p)) (leaves p) optr) H.
```

Here `leaves` extracts the leaves of a `ptree`. Bornat *et al.* [3] choose to use classical conjunction to capture these two views of the tree. We found it more convenient to prove an equivalence and forgo the conjunction, since most operations do not use the leaf list and it reduces the burden of proving additional facts about the leaf-trunk division. In addition to the heap shape described by `repTree`, the model must satisfy the appropriate sortedness constraints. Recall that this consists of an ordering on the elements in a node and between the keys and subtrees as shown in Figure 3. More formally, the constraints are captured by the predicate `treeSorted`:

$$\begin{array}{l}\texttt{treeSorted}\ 0\ min\ [(k_1, -), ..., (k_n, -)]\ max \iff \\ \quad min < (\texttt{Key}\ k_1) \wedge k_1 < k_2 \wedge ... \wedge (\texttt{Key}\ k_n) \leq max\end{array}$$

$$\begin{array}{l}\texttt{treeSorted}\ (h+1)\ min\ [(k_1, t_1), ..., (k_n, t_n)]\ max \iff \\ \quad \texttt{treeSorted}\ h\ min\ t_1\ (\texttt{Key}\ k_1) \wedge \\ \quad \texttt{treeSorted}\ h\ (\texttt{Key}\ k_1)\ t_2\ (\texttt{Key}\ k_2) \wedge ... \wedge \\ \quad \texttt{treeSorted}\ h\ (\texttt{Key}\ k_{n-1})\ t_n\ max\end{array}$$

where min and max are drawn from the inductively defined type:

```
Inductive exkey: Set :=
| Min: exkey | Key: key -> exkey | Max: exkey.
```

which provides minimal and maximal elements, and where key comparisons are lifted to `exkey` in the obvious way.

Finally, to satisfy the finite map interface of Figure 2 we must define an abstract type `handle` that abstracts the details of the B+ tree implementation and hides the internal definitions of `ptree`s and their associated properties. The abstract `rep` predicate relates B+ tree handles and key-value association lists in a given state.

We begin by defining a function `as_map` which extracts an association list of keys and values from a `ptree`:

```
Fixpoint as_map (h : nat) : ptree h -> AssocList :=
  match h as h return ptree h -> AssocList with
    | 0    => fun m => snd m
    | S h' => fun m =>
      List.flat_map (fun x => as_map (snd x))
        (contents m) ++ as_map (next m)
  end.
```

Finally, we define a `handle` to be a pointer to a pair, where the first component is the root of the B+ tree, and the second is the irrelevant `ptree` model for the B+ tree. These facts, along with the shape and sortedness invariants, are captured in the definition of the `rep` predicate, which is similar to the definition below:

```
Definition handle : Set := ptr.

Definition rep (hdl : handle) (m : AssocList)
  : heap -> Prop :=
  Exists (r : ptr) (h : nat) (p : ptree h),
    hdl --> (r, existT (fun h => [ptree h]) h p) *
    repTree h r None p *
    [m = as_map p] * [treeSorted h p Min Max].
```

In this definition, we have taken some notational liberties by using "`Exists`" to stand for heap-dependent existential quantification and "`-->`" to represent the points-to relation. By packing the `ptree` model with the actual B+ tree, we avoid the need to search for a model during proofs. Rather, as the B+ tree is updated, we perform the corresponding (functional) updates on the associated model. This sort of "ghost state" is often useful for simplifying Hoare-style proofs.

An alternative to packaging the B+ tree with its model is to simply show that there is at most one `ptree` that a given pointer and heap can satisfy (i.e., that `repTree` is *precise* [4, 17]). However, this is complicated by the fact that the types of `ptree`s are indexed by height, and thus comparison (and substitution) demand the use of heterogenous, "John Major" equality [10].

6.2 Implementation of B+ Tree Operations

In order to meet the finite map specification, we must provide the five operations given in Figure 2. The implementations of `new` and `free` are relatively straightforward.

Both `lookup` and `insert` recurse over the tree to find the leaf which should contain a given key, perform the appropriate action

on the leaf, and merge the result into the tree. Sexton and Thielecke [24] formulate this by building a language of tree-operations for a stack-machine. We take a similar approach by factoring out the steps of finding the appropriate leaf, performing an operation, and then merging the results. This is encapsulated in the `traverse` command which takes an operation `cmd` to apply at the leaf. The definition of `traverse` is sketched in Figure 4 as ML code.

```
let traverse tKey tree cmd =
  let (v,ms) = trav tkey !tree cmd in
  (tree := match ms with
           | Merge tr'   -> tr'
           | Split l k r ->
             buildTree ((k,l)::nil) r
         end) ; v

let rec trav tKey tree cmd =
  if isLeaf tree then
    cmd tree
  else
    let tr = findSubtree tKey tree in
    let (v,ms) = trav tKey tr cmd
    (v,
     match ms with
     | Merge tr' -> replaceSubtree tKey tree tr'
     | Split l k r -> spliceSubtree tKey tree l k r
     end)
```

Figure 4. The `traverse` function for performing leaf-local tree computations.

A call to `traverse` invokes a helper operation `trav` which searches for the appropriate leaf using `tKey` to guide the search and then executes `cmd` on that leaf. The execution of `cmd`, or a recursive call to `trav`, results in a value `v` and either a new tree (`Merge tr'`) or a pair of trees `l` and `r` and a key `k` (`Split l k r`) such that `k` is greater than or equal to the elements of `l` and less than the elements of `r`. In the former case, we simply replace the original sub-tree with the newly returned tree. In the latter case, we must splice the two trees into the current node or, at the top-level, build a fresh node (increasing the height of the tree by one.) The `spliceSubtree` operation must replace the old sub-tree `tr` with the new sub-trees `l` and `r`. In the easy case, we have room in the interior node's array, but must shift some elements over to make room. In the hard case, we may not have room and thus must split the node in half and return a pair of trees to the caller. The key point is that `traverse` is general enough to support a range of operations, including both `lookup` and `insert`, while maintaining the B+ tree invariants.

Of course, our Ynot code demands considerably more annotation to capture the specification of the `cmd` argument, and to reflect the appropriate invariants for recursion, as well as the proofs that the code respects pre- and post-conditions. For example, to describe the post-condition for `cmd`, we abstract over a return type T for the value returned by `cmd` and predicate Q of type:

$$Q : \texttt{AssocList} \rightarrow T \rightarrow \texttt{AssocList} \rightarrow \texttt{heap} \rightarrow Prop$$

The predicate is intended to capture the effect of executing `cmd` by relating the association list model before and after the command is executed, as well as the value returned by `cmd`. For example, we can implement `lookup` by using a command with post-condition specified by:

$$Q_{lookup}\ mpre\ rt\ mpost = \\ [mpost = mpre] * [rt = \texttt{specLookup}\ tKey\ mpre].$$

Here, the model (i.e., list of key-value pairs in the leaves) remains unchanged, and the return value is equivalent to performing a functional lookup on the initial association list. We can implement `insert` by using a command with post-condition specified by:

$$Q_{insert}\ mpre\ rt\ mpost = \\ [mpost = \texttt{specInsert}\ tKey\ tValue\ mpre] * \\ [rt = \texttt{specLookup}\ tKey\ mpre].$$

This captures the fact that the new model is obtained from the old by inserting the key-value pair into the association list and that the returned value is equal to the result of looking up the key in the original model.

In addition, we need knowledge that the effect of running `traverse` (and hence `trav`) on the tree will only affect the path from the root to the leaf of interest. That is, we must show that Q satisfies a frame property with respect to the rest of the model:

$$\forall H\ min\ min'\ max'\ max\ tKey\ low\ i\ i'\ hi\ rt, \\ min' < tKey \leq max' \rightarrow \\ sorted\ min'\ i\ max' \rightarrow sorted\ min'\ i'\ max' \rightarrow \\ sorted\ min\ low\ min' \rightarrow sorted\ max'\ hi\ max \rightarrow \\ Q\ i\ rt\ i'\ H \rightarrow Q\ (low{+\!\!+}i{+\!\!+}hi)\ rt\ (low{+\!\!+}i'{+\!\!+}hi)\ H$$

The frame properties for Q_{lookup} and Q_{insert} follow from the fact that `specLookup` and `specInsert` do not alter portions of the tree which do not intersect the target key.

The abstraction provided by `traverse` considerably reduces the burden of writing and verifying both `lookup` and `insert`, since we only need to define the operation to perform at the leaf. Furthermore, we believe that `traverse` will be useful for defining other operations which traverse the B+ tree. One such simple operation is insertion without replacement in a single traversal.

The final interface command is iteration. Verifying the implementation of `iterate` relies crucially on our ability to view a B+ tree in two ways, as per theorem `repTree_iff_repTrunk`. Conceptually, the implementation can be broken into 4 steps:

1. Follow left links in the tree until the first leaf is reached.
2. *Change views, separating the heap into trunk and leaves.*
3. Iterate over the "skirt," calling a function on each element in each leaf.
4. *Change views, recombining the leaves and trunk.*

Step 1 is a simple recursion of the tree which could be eliminated if the head of the skirt was stored along with the root. Steps 2 and 4 are purely proof-associated steps, incurring no run-time overhead, and follow immediately from the `repTree_iff_repTrunk` theorem defined previously. Step 3 is a straightforward, nested iteration over a linked-list of arrays.

7. The Storage Manager

The RDBMS maintains an environment of tables in memory, and persistence is implemented by serializing the relations as strings and writing/reading these strings from disk using inverse functions. A set of saved tables can be loaded with a user command, and the relation information is checked and constructed during loading. Verification amounts to checking an internal consistency property that ensures reading and printing are appropriate inverses:

```
PrintTable: forall I, Relation I -> string
ReadTable : forall I, string -> option (Relation I)
Theorem storage_ok : forall I (tbl: Relation I),
  ReadTable (PrintTable tbl) = Some tbl.
```

8. Evaluation

Overall, we found this project to be extremely challenging, as it required a wide range of formalization tasks, from encoding semantics and compiler correctness, to reasoning about pointer-based data structures, to issues involving parsing and serialization. On the one hand, it is impressive that Coq supports such a wide range of tasks and, furthermore, provides the abstraction capabilities to allow for such a modular decomposition. On the other hand, we found the difference between informal models and invariants and their Coq representations to be much larger than one would hope.

It is instructive to begin by comparing the amount of different kinds of code that make up the development. Figure 5 describes the breakdown of functional code, imperative code, and proofs in our RDBMS. The "Functional" column gives line counts for both pure code (including specifications) and functional code used at run-time. The "Imperative" column gives line counts for Ynot code, which is written in a style that generates a large number of verification conditions. The "Proofs" column gives line counts for both tactics and Coq terms used to construct proofs.

	Functional	Imperative	Proofs
Model	360	0	700
SQL Compiler	840	0	440
SQL Engine	0	250	1350
B+ tree	360	510	5190
Storage	450	160	340
Total	2110	920	8020

Figure 5. Numbers of lines of different kinds of code

These totals do not include our FSet modifications and the base Ynot tactics and data structures that we use (such as the packrat parser toolkit). They do include the [de]serializers for basic types and the grammars used for parsing queries and tuples. Line counts were taken "as-is", and there is no doubt that we can improve these. Nevertheless, the numbers indicate that formalizing a system to this degree requires a substantial investment over and above the code. On the other hand, many of the components are directly re-usable in other contexts (e.g., the finite map interface and its associated theory, the B+ trees, etc.).

8.1 Some Lessons Learned

In what follows we highlight some of the lessons learned from our development. Many of these lessons are well known to Coq experts, but we hope this discussion will help those interested in similar developments.

There are a number of design decisions we made that initially seemed "right" but ultimately led to complicated proofs. For example, it took many tries to find the right formulation of the invariants for the B+ trees. Originally, we avoided introducing the intermediate `ptree` models described in Section 6.1. Instead, we tried to define predicates that would directly connect association lists with the pointer graph in the heap. However, this definition required a large number of existential quantifiers (up to 25 in many goals). This made reasoning difficult, as we were forced to provide witnesses explicitly. Introducing `ptree`s and explicitly associating them with the B+ tree as ghost state helped avoid this complication.

In general, we found that one should avoid disjunctions of any flavor when something can be readily computed. This is because Coq will automatically reduce computations but requires explicit guidance to eliminate disjunctions. For example, we originally encoded `ptree`s as an inductively defined type:

```
Inductive ptree (h:nat) : Set :=
| Leaf : ptr -> list (key * value) -> ptree 0
| Node : forall h, ptr -> list (key * ptree h) ->
                      ptree h -> ptree (S h).
```

However, eliminating such definitions in proofs requires an explicit use of inversion, even when the height is known to be zero or non-zero. In contrast, Coq automatically reduces the recursive definition based upon `match`. In many cases, this substantially reduced the number of manual steps needed in proofs.

Whether and how to use dependent types in our definitions was another source of frustration. On the one hand, we found dependency useful to capture schemas for relational operations and to rule out various error cases that would otherwise arise. On the other hand, writing transformations, such as the optimization to fuse adjacent projections, requires tedious arguments and coercions to ensure that the output is well-typed. Newer languages such as Epigram [11] and Agda [16] provide better support for programming with dependent types, and Matthieu Sozeau is working to adapt many of these ideas to the Coq setting [26], so we are hopeful that this frustration will diminish over time.

Another lesson we have learned is the value of automation, both through hints and custom tactics. This insulates proofs against code changes and greatly speeds up the proof process as tactics mature. In our previous work on Ynot [5], we advocated a style of proof (derived from our colleague Adam Chlipala) where instead of writing many small, manual proof steps, it is better to write a custom tactic that searches for a proof of the goal. Our experience with this project confirms the usefulness of this methodology. For example, the proof terms for the B+ tree are about 22MB in size. Writing them by hand would be impossible, but writing tactics that search for these terms has proven remarkably robust throughout the evolution of our code.

Another lesson we have learned is that the Coq modules are useful for controlling name spaces, but their second-class nature makes it difficult to use them effectively for abstraction. For instance, the section mechanism of Coq, which is extremely useful for factoring parameters, does not work well in the context of modules: we cannot define a module within a section. Rather, we found core language mechanisms, such as dependent records and type classes, to be more useful than modules. Consequently, we avoided sophisticated use of the module system when possible.

8.2 Related Formalization

A final lesson regards the formalization of relational algebra. The idea of mechanizing (data models more general than) relational algebra goes back to at least NuPrl [21]. The inspiration for our work, however, is the formalization of the relational algebra in Agda found in Carlos Gonzalia's Ph.D. thesis [9]. Like that thesis, we use axiomatic finite sets; however, we opted for a more concrete tuple representation and a different, but equivalent, choice of base relational operations.

The thesis (essentially) represents a schema as a function from a finite set of columns to an Agda type.

```
Inductive Fin : nat -> Set :=
| zero : forall k, Fin (S k)
| succ : forall k, Fin k -> Fin (S k).

Definition Schema (k: nat) : Set := Fin k -> Set.
Definition Tuple k (I: Schema k) : Set :=
  forall (col: Fin k), I col.
```

We originally adopted this representation, but found that the easiest way to establish certain results like the decidability of equality over schemas was to convert schemas into lists and use an isomorphism between the two representations. Other times this representation

forced us to appeal to functional extensionality ($\forall x, fx = gx \rightarrow f = g$), which is not a theorem in Coq. Using this axiom complicates reasoning since axioms do not have computational behavior. Finally, while induction on k is possible, our more concrete definition provides a direct induction principle that simplifies proofs.

We also differ from the thesis in our choice of relational basis operations. We implemented cartesian product and included a permutation operation to simplify SQL compilation. Instead of product, the thesis implements join and supports projection through two operations for splitting a relation in half. Although in principle the two models are equivalent, certain operations possible in SQL, like swapping two columns, are easier to encode in our basis.

In [19], Oury and Swierstra give a dependently-typed relational algebra syntax that is similar to ours, but that uses column names instead of numbers. Their approach leads to a different proof burden; for instance, in their approach the schemas A and B must have disjoint attribute names in $A \times B$. We expect to study this particular tradeoff more closely in the future.

9. Future Work

Our work lays the foundation for mechanically verified RDBMSs, but to be practically useful for real systems, there are a number of tasks that remain. Some of these tasks are relatively modest extensions to the current implementation. For example, finishing incorporating key information to enable efficient point and range queries. This, along with additional relation statistics such as size and selectivity measures, will enable additional optimizations such as better join planning and more semantic optimizations. Additional B+ tree features needed for range queries will also enable more efficient relational algorithms such as merge-join. Finally, reifying the low-level query plan will allow us to fuse operations to avoid materializing transient data. With these features, and support for aggregation and basic transactions, the system should be usable for simple in-memory, single-threaded applications.

In the long term, realizing the ACID guarantees of concurrency, transactional atomicity and isolation, and fault-tolerant storage will demand substantial extension. For example, Ynot does not yet support writing or reasoning about concurrent programs, though we have done some preliminary work extending Hoare Type Theory with support for concurrency and transactions [13] based on the ideas of concurrent separation logic [4]. We expect that implementing and verifying the correctness of high-performance, concurrent B+ trees will be a particularly challenging problem. As another example, we need to find the right model for disk I/O – which accurately reflects possible failure modes – and incorporate this into the program logic. Our ultimate goal is to combine these features with the enhancements above to obtain a fully verified, realistic RDBMS that can be used in safety- and security-critical settings.

Acknowledgments

We would like to thank Adam Chlipala and Matthieu Sozeau for their help with proof engineering in Coq. We would also like to thank David Holland and Margo Seltzer for bringing perspective from the database community.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Database Foundations*. Addison-Wesley, 1995.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] R. Bornat, C. Calcagno, and P. OHearn. Local Reasoning, Separation and Aliasing. *Proc. SPACE*, volume 4, 2004.

[4] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[5] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proc. ICFP*, 2009.

[6] C. J. Date. *Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison Wesley, 2006.

[8] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *Proc. SIGMOD*, 2001.

[9] Carlos Gonzalia. *Relations in Dependent Type Theory*. PhD Thesis, Chalmers University of Technology, 2006.

[10] Conor Mcbride. Elimination with a motive. In *Proc. TYPES*, 2000.

[11] Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004.

[12] James Mckinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in epigram. In *J. Functional Programming*, 2006.

[13] Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards type-theoretic semantics for transactional concurrency. In *Proc. TLDI*, 2009.

[14] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proc. ICFP*, 2006.

[15] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *Proc. ICFP*, 2008.

[16] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[17] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[18] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. CSL*, 2001.

[19] Nicolas Oury and Wouter Swierstra. The power of pi. *Proc. ICFP*, 2008.

[20] Sharon E. Perl and Margo Seltzer. Data management for internet-scale single-sign-on. In *Proc. WORLDS*, 2006.

[21] P. Rajagopalan and C. P. Tsang. A generic algebra for data collections based on constructive logic. In *Algebraic Methodology and Software Technology*, volume 936 of *LNCS*, pages 546–560. Springer Berlin / Heidelberg, 1995.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, 2002.

[23] Alan Sexton and Hayo Thielecke. Reasoning about b+ trees with operational semantics and separation logic. *Electron. Notes Theor. Comput. Sci.*, 218:355–369, 2008.

[24] Alan Sexton and Hayo Thielecke. Reasoning about b+ trees with operational semantics and separation logic. *Electron. Notes Theor. Comput. Sci.*, 218:355–369, 2008.

[25] Carsten Sinz. System description: Ara - an automatic theorem prover for relation algebras. In *Proc. CADE-17*, 2000.

[26] Matthieu Sozeau. Program-ing finger trees in coq. In *Proc. ICFP*, 2007.

[27] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proc. TPHOLs*, 2008.