Paper Title:	Query Optimizer Correctness: Applying Formal Methods to Query Planning
Reference Number	413
Authors	Mitch Cherniack and Xiaoyu Wang (No author serves on any VLDB 2002 PC)
Contact Author	Mitch Cherniack MS 018, Department of Computer Science Brandeis University Waltham, MA, 02454 781-736-2738 (voice) 781-736-2741 (fax)
Topic Area	Core Database Technology
Category	Research
Relevant Topics	Advanced Search, Query, and Approximation; Database Languages; Manageability of Database Systems

Query Optimizer Correctness: Applying Formal Methods to Query Planning

Mitch Cherniack
Brandeis University
Waltham, MA 02454
mfc@cs.brandeis.edu

Xiaoyu Wang
Brandeis University
Waltham, MA 02454
wangxy@cs.brandeis.edu

Abstract

As very large and complex database components, query optimizers are brittle and error-prone. Over the past several years, testing has revealed that database systems frequently return the wrong results to queries. Although it can be shown that testing is sound (i.e., that identified bugs are indeed bugs), it is difficult to prove that testing is complete (i.e., that it finds all bugs).

Testing is a form of software validation. In this paper, we present work addressing the verification, rather than validation, of query optimizers. This work introduces formal methods to the design and construction of query optimizers, thereby ensuring that errors are never introduced into optimizer software in the first place. Specifically, in this paper we present the formal specification of a generic *plan algebra* (GPA), as well as the *files of records* it acts upon and the supporting structures (e.g., *indexes*) it uses. We then show the viability of this algebra by describing translations of standard SQL queries into plans, and some verified plan generation rules.

1 Introduction

Query optimizers (optimizers) map queries to executable plans. An optimizer is *correct* if it maps every query to a plan that returns the data specified by the query. This paper presents work contributing to our goal of designing and implementing an *optimizer generator* that will assist developers and researchers in the design and implementation of *provably correct* query optimizers.

The motivation for this work comes from the recognition that optimizers are complex and error-prone software systems. In his keynote address at SIGMOD '98, Bill Gates reported that testing had revealed hundreds of bugs in the query processor for SQL Server [10]. At the 1998 NSF Workshop on Industrial/Academic Cooperation in Database Systems, Paula Hawthorne commented that the database software industry is faced with a correctness crisis much like the safety crisis that faced the 1960's auto industry [19]. Hawthorne concludes that database software development must take a more formal approach to make results more robust. The correctness issue affects not only optimizer development but optimizer research also; the "COUNT bug" was one of a number of errors revealed by Kiessling [21] of optimization techniques for relational databases that were published by Kim two years before [22]. The emergence of more complex data (e.g., object-oriented) and queries (e.g., decision support) since Kiessling's report has only exacerbated the problem of ensuring query optimizer correctness.

In earlier work [6, 4, 5], we built a *query rewriter generator* (COKO-KOLA) whose inputs could be proved correct with an automated theorem prover. In this paper, we present complementary work towards the development of a *query planner generator* with the same correctness properties. A *query planner* constructs *query plans*: compositions of algorithms (e.g., hash join) and access methods (e.g., index scans) whose execution generates the result to a query. In the spirit of COKO-KOLA, our query planner generator will accept a formal specification of a query planner as input, and use it both to generate a query planner and to verify its correctness. In this paper, we take the first step towards this goal by presenting a formal specification of a generic plan algebra. This approach parallels the approach we took before, where the formal specification of a logical query algebra (KOLA) [6] preceded the design of the query rewriter generator.

The rest of this paper proceeds as follows. We begin in Section 2 presenting background in formal methods and query optimization. We also describe how existing optimizer generator tools differ from the one we propose, both in objective and design. In Section 3, we present formal specifications of files of records (our assumed underlying representation of relations), and relevant properties of files (e.g., sort order or grouping) that can be inferred and exploited by the optimizer to improve the plans they generate. We present our generic plan algebra (GPA) and its formal specification in Section 4. In Section 5, we show the viability of this algebra by translating common SQL queries into plans expressed with its operators, and in Section 6, we demonstrate some query planning rules that we have verified with a theorem prover. We describe related work in Section 7, and summarize and discuss future work in Section 8.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

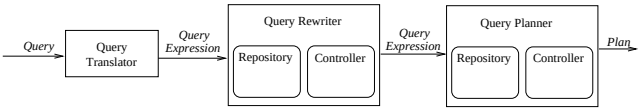

Figure 1: A Typical Architecture for Optimizers

2 Background

Formal methods are mathematically-based languages, techniques and tools for specifying and verifying complex software systems [7]. Formal methods tools include: a *specification language* in which the semantics of a system can be expressed, and an *automated theorem prover* that assists developers in proving certain properties about their specified system (e.g., that the implementation is faithful to the specification).

Software engineers introduced *formal methods* to assist with the verification of complex software systems. Systems communities have been slow to adapt formal methods techniques, in part because many of the early formal methods tools were difficult to use, and also because of the perception that their use added prohibitive cost to the software development process [18]. But the newer generation of formal methods tools are far easier to use than their predecessors – the Larch [15] specification tool and theorem prover, for example, require only a basic knowledge of discrete mathematics. And formal methods tools are now viewed as a way to reduce the long-term costs of maintaining and debugging software systems. As Jim Gray points out in his 1999 Turing Award address, 50% of development costs are now spent maintaining and debugging code, making software the only commodity in cyberspace that is getting more expensive and less reliable [14]. Finally, the cost of applying formal methods tools can be reduced by targeting the *safety-critical* components of systems: those components that alone determine the correctness of the system. Formal methods have been successfully applied to the development of such systems as transaction processing systems [20], compilers [16], real-time systems [27], [28], and network management systems [9]. Query optimizers represent another natural application of this technology.

2.1 Safety-Critical Components of Optimizers

To help identify the safety-critical components of query optimizers, we show a generic query optimizer architecture in Figure 1. This architecture consists of the three components described below.

The Query Translator: The input to a query optimizer is a query expressed in a user-level query language such as SQL. This query first gets gets submitted to a query translator that translates the query into an internal representation (such as one based on the relational algebra). We use the term *query expression* to refer to a query's internal representation.

The Query Rewriter: The query rewriter rewrites a query expression into an equivalent query expression that is likely to lead the optimizer to consider alternative plans.[1] For example, query rewrites that unnest a nested query free an optimizer to consider join-processing strategies rather than being limited to nested loop algorithms. Optimizers use query rewriting in one of two ways. Some (e.g., Starburst [29]) rewrite a query expression into a single, equivalent query expression that by some heuristic metric is considered "better", and submit this expression to the query planner. Others (e.g., Cascades [12]) generate multiple equivalent query expressions during query rewriting, leaving the decision to the query planner as to which results in the best plan.

The Query Planner: The query planner rewrites a query expression into a *plan*; an algorithm expressed in terms of operators designed for efficient data management (e.g., index scans, sort-merge joins and hash joins). This process typically involves generating a set of alternative plans for the query expression, and choosing one that is best according to cost estimates.

The correctness of a query optimizer depends upon the correctness of its translator, query rewriter and query planner components. A translator is correct if translation results in a query expression with the same semantics as the original query. A query rewriter is correct if every query gets rewritten to a query with the same semantics. A query planner is correct if the result of executing a plan is the data specified by the query expression from which it was generated.

The query rewriter and query planner can be further subdivided to isolate their safety-critical components. (All of the translator is safety-critical.) Figure 1 subdivides both components into *repositories* and *controllers*. A repository describes all possible ways that queries can be rewritten into other queries or plans. A repository consists of individual *query rewrites* or *plan generating rewrites* that rewrite some subset of queries in uniform fashion. Controllers are responsible for choosing one or more rewrites in the repository to apply to (*fire on*) a given query expression. In many systems, the distinction between repository and controller is purely conceptual. Indeed, it is difficult to define these components independently of one another as effective controller design requires understanding the specific rewrites at its disposal. But the distinction, even at only a conceptual level, is crucial for identifying the safety-critical components of the optimizer. Specifically, the rewrites contained in repositories are safety-critical, whereas controllers are not. Intuitively, this is because an optimizer that only has a set of correct rewrites that it can fire cannot be made incorrect by a controller that is constrained to choose from this set. Thus, the safety-critical components of the optimizer are those that rewrite query expressions into alternative query expressions or alternative representations: the query translator, the query rewriter repository and the query planner repository.

2.2 Optimizer Generators

Optimizer generators were first proposed in the mid-1980's as a means of achieving *extensibility* in query optimizer development. The first optimizer generator was Exodus [2] (which later evolved into Volcano [13] and Cascades [12]) which introduced the idea of specifying

[1] Because they process algebraic query expressions, many systems refer to query rewriting as *algebraic query optimization*.

an optimizer in terms of a set of *rules*. This made optimizers extensible, as one only had to modify a set of rules and regenerate an optimizer to change its behavior. Other rule-based systems with similar extensibility goals followed, including Starburst [17], Gral [1], EDS [8] and OGL [30].

Beyond extensibility, rule-based optimizer generators are ideal for optimizer verification. Rules in a rule-based optimizer specify minor modifications of query expressions that preserve semantics. Thus, the process of proving a rule-based optimizer correct simply requires proving each individual rule to be correct. But the rule-based systems listed above are ill-suited for generating verifiable optimizers, as rules in these systems get expressed in part or completely with code. It is difficult to manually prove that code performs a given task, let alone with an automatic theorem prover. In fairness, formal verification was not a stated objective for any of these systems. Nonetheless, a different approach to expressing rules is necessary, as rules comprise the safety-critical components of optimizers (i.e., the repositories). The rest of this paper describes our work in expressing query planner repositories with a formally specified plan algebra.

3 Formal Specifications

The purpose of a query planner specification is:

1. to specify the desired behavior of a query planner that will be constructed by the query planner generator, and
2. to enable verification of the plan generating rewrites it executes.

The first step towards making query planners specifiable is to formally specify a plan algebra so that optimization rules expressed in terms of this algebra have some basis for determining correctness. We are concerned with two kinds of rules:

A *Translation Rule* translates a query expression into a plan. An example translation rule is one that translates a logical join query ($A \bowtie B$) into a plan that performs a merge join after sorting the files storing the contents of A and B.

A *Refinement Rule* rewrites one plan into another, presumably cheaper plan, perhaps depending on satisfaction of certain conditions. An example refinement rule is one that rewrites a plan that performs a merge join after sorting its inputs, into a plan that sorts only one of its inputs on the condition that the other input is already sorted in the appropriate way.

In Section 4, we present a formal specification of our generic plan algebra (GPA). In this Section, we present the formal specifications of the entities it acts upon: 1) files of records (assumed to be the way that relations are physically represented) and 2) indexes. The formal specifications have been defined in the Larch algebraic specification language, LSL [15].

Example LSL specifications are shown in Figures 2 and 3. Larch specifications defines *traits*, the Larch equivalent of *abstract data types* [25]. The set trait is parameterized by the type parameter, `T`. Therefore, the set specification of Figure 2 specifies the semantics of any homogeneous set of objects of type `T`. A trait has two clauses: the `introduces` clause contains the signatures of all operators associated with the trait, and the `asserts` clause contains the axioms that define them. The set trait of Figure 2 includes the standard set constructors $\emptyset$ (empty set) and `insert`, and defines set membership ($\in$) and set equality ($=$) in terms of sets expressed in terms of these constructors.[2] Note that `Bool` is a built-in trait for LSL (as are `Int`, `String` etc.). The predicate trait of Figure 3 defines predicate properties, `preord` (*preorder*), `parord` (*partial order*) and `sortord` ("*sort order*") (which refers to a predicate that can be used to sort a file).[3]

```
Set [T]: trait
introduces
 ∅ :                              →  Set [T]
 insert :          T, Set [T]     →  Set [T]
 _ ∈ _ :           T, Set [T]     →  Bool
 _ = _ :     Set [T], Set [T]     →  Bool
asserts
 ∀ x, y: T, A, B: Set [T]
  ¬ (x ∈ ∅)
  x ∈ insert (y, A)  ⟺  (x = y) ∨ (x ∈ A)
  A = B              ⟺  (x ∈ A ⇔ x ∈ B)
```

Figure 2: An LSL Specification of `Set`

3.1 Files of Records

To specify the semantics of GPA, we must specify each operator's inputs and outputs. For this purpose, we assume that relations are sets of tuples[4] and are represented by files of records (*files*), and that files are the arguments to and objects returned by plan operators. It is important to note that our specification does not require files to be materialized. For example, the result of a subquery might be a "file" whose records are not stored but instead pipelined to other operations, as in a filtered scan (`fscan`) that returns a subset of the records of the file it scans.

3.1.1 Files

File specifications must have the following properties:

1. To support the verification of translation rules, it is necessary to characterize the *contents* of a file

[2] Note that `insert` is not an operation but a constructor: the set, $\{1, 2, 3\}$ would be expressed in this notation as "`insert (1, insert (2, insert (3, ∅)))`.".

[3] Reflexivity, symmetry, antisymmetry and transitivity are defined in the usual way: p is `reflexive` if $p(x, x)$ for all x; `symmetric` if $p(x, y) \Rightarrow p(y, x)$ for all x and y, `antisymmetric` if $p(x, y) \wedge p(y, x) \Rightarrow x = y$ for all x and y, and `transitive` if $p(x, y) \wedge p(y, z) \Rightarrow p(x, z)$ for all x, y, and z.

[4] As was shown in [3], allowing for duplicates in relations (i.e., modeling relations as *bags* rather than *sets* of tuples) is a straightforward extension to the formal specification of query operators, though correctness proofs (which must account for duplicates in query results) are made more tedious as a result. While our work assumes relations as bags, we simplify presentation of the work here by showing a set-based model.

```
Predicates [T]: trait
introduces
 preord, parord, sortord: (T, T → Bool) → Bool
asserts
 ∀ x, y: T, p: (T, T → Bool)
  preord(p) ⟺ reflexive(p) ∧ transitive(p)
  parord(p) ⟺ preord(p) ∧ antisymmetric(p)
  sortord(p) ⟺ preord(p) ∧ ¬ symmetric(p)
```

Figure 3: An LSL Specification of `Predicates`

```
File [T]: trait
introduces
 empty                          → File [T]
 file: Set [T], OrdCon [T]      → File [T]
 ordof: File [T]                → OrdCon [T]
 setof: File [T]                → Set [T]
 __ = __: File [T], File [T]    → Bool
asserts
 ∀ A: Set [T], F, F′: File [T], o: OrdCon [T]
  o = ordof(F) ⟺ ∃ A (F = file(A, o))
  A = setof(F) ⟺ ∃ o (F = file(A, o))
  F = F′ ⟺
   (setof(F) = setof(F′) ∧ ordof(F) = ordof(F′))
```

Figure 4: An LSL Specification of `File`

of records to see if the plan resulting from translation produces the same data as that specified by the query expression that was translated.

2. To support the verification of refinement rules, it is necessary to be able to characterize the order in which records in the file are stored.

To support the verification of translation and refinement rules, we specify files as pairs, (S, o) such that S is the relation (set) represented by the file, and o is an *ordering constraint* that specifies the order in which records in the file appear. (Ordering constraints are discussed in Section 3.1.2.) Given this representation, the proof of any translation rule translating query Q to plan P simply requires showing that for any file, F denoting some relation, R, that $Q(R) = \texttt{setof}\ (P(F))$, such that `setof` returns the *contents* of a file. The proof of any refinement refining plan P_1 into plan P_2 simply requires showing that for any file, F denoting some relation, R, that $P_1\ (F) = P_2\ (F)$.

The Larch formal specification of files is shown in Figure 4. Operators on files include a constructor, `file`, that accepts a set with schema type `T` and an ordering constraint, and constructs a file of records of schema type `T`. Also defined in the specification are `setof` (described earlier), `ordof`, which returns the ordering constraint associated with a file, and "__ = __" (equality on files). Note that two files do not have to be the *same* file to be equal: they need only have the same contents and the same ordering.

3.1.2 Ordering Constraints

The verification of a translation rule requires showing the equivalence of set expressions: one being the logical query expression that is translated, and the other being the set component of the file expression into which it is translated. Theorem prover verification of set equivalence is discussed at length in [6] and no further here.

The verification of a refinement rule is more subtle. For some plans, correctness of refinement only requires set equivalence (e.g., plans for which ordering of the result is irrelevant). On the other hand, for queries requiring ordered results, or plans that are *subplans* (i.e., whose results are fed to other plan operators), order preservation is crucial. Consider, for example, a subplan that produces results in sorted order in preparation for a merge join. Such a plan cannot be replaced with another plan that ignores order or produces an ordering on some attribute not relevant to the join.

Thus, we base the correctness of a refinement rule on how it preserves both set and order equivalence. Note that order equivalence of plans **need not** guarantee that records in the results of executing these plans be in exactly the same order. Consider, for example, alternative plans, P_1 and P_2, for a query that orders records in a `Person` relation (Figure 5a) by `Age` (in ascending order). Suppose that when P_1 is applied to `Person`, it returns the relation shown in Figure 5b and that when P_2 is applied to `Person`, it returns the relation shown in Figure 5c. The relations of Figure 5b and 5c differ – `Joe` and `Bill` appear in a different order in the two results. However, both results are acceptable results to this query. Therefore, it should be acceptable to refine P_1 into P_2 (or vice-versa) to process this query. Similarly, consider plans that group relation `Person` by `Dept`. In this case, any plans that return either of the results shown in Figures 5d or 5e or any of the 22 permutations of these results that vary according to the ordering of `Helen` and `Fred` (`Eco`), or of `Tom`, `Joe` and `Bill` (`CS`) should be considered acceptable.

This "loose" notion of ordering and grouping makes it possible to specify query planners that are free to consider a wide range of plans to satisfy a given query. But more than this, knowledge of ordering and grouping of files makes it possible to express *conditional* refinement and translation rules that depend on these properties. For example, consider a plan that must remove duplicates from the result of a query. A common duplicate elimination algorithm involves first partitioning a relation based on a hash function on entire tuples. A hash index is then constructed from the partitions, during which time duplicate tuples are recognized and discarded [24]. This expensive process can be circumvented if the result of a query is already grouped according to some set of attributes comprising a key. In this case, a simple one pass scan is required that retains one tuple from each group. This duplicate elimination is not only less expensive than the hash-based implementation, but also doesn't require prior materialization of the relation. Similarly, if a query planner knows that the records contained in one a file are ordered on a particular attribute, it might choose to translate a join of two relations into a merge join, recog-

Name	Age	Dept
Joe	34	CS
Helen	25	Eco
Fred	38	Eco
Bill	34	CS
Tom	27	CS

(a)

Name	Age	Dept
Helen	25	Eco
Tom	27	CS
Joe	34	CS
Bill	34	CS
Fred	38	Eco

(b)

Name	Age	Dept
Helen	25	Eco
Tom	27	CS
Bill	34	CS
Joe	34	CS
Fred	38	Eco

(c)

Name	Age	Dept
Helen	25	Eco
Fred	38	Eco
Tom	27	CS
Joe	34	CS
Bill	34	CS

(d)

Name	Age	Dept
Tom	27	CS
Bill	34	CS
Joe	34	CS
Fred	38	Eco
Helen	25	Eco

(e)

Figure 5: Table `Person` (a) and 4 Permutations (b-e)

nizing that at least one of the sort operations required as a preprocessing step is unnecessary.

The grouping and ordering characteristics of a file are specified in an ordering constraint, whose specification is shown in Figure 6. Ordering constraints have two constructors: `none` and `ord`. `None` is associated with a file either when the file's records are unordered, or when no ordering information is known. `Ord` takes a preorder predicate (see Figure 3) over pairs of objects of type `T` and an ordering constraint as arguments, and returns another ordering constraint.

An ordering constraint specifies two kinds of relationships between records. Given records, x and y, we write:

`x` $\prec$ `y WITH o`, if x *precedes* y, and
`x` $\equiv$ `y WITH o`, if x and y are *grouped together*

in any file with ordering constraint, `o`. Consider again, relation `Person` of Figure 5(a). Suppose that the file storing this relation is specified as

`File (Person, ord (leqAge, none))`,

such that the predicate, `leqAge`, is true of two `Person` records, x and y iff x.`Age` $\leq$ y.`Age`. This says that for any records in the file, x and y:

$x \prec y$ `WITH ord (leqAge, none)` $\iff$ x.`Age` $\leq$ y.`Age`.

Thus, it must be the case that `Helen` precedes all other records in the file, that `Tom` precedes `Joe`, `Bill` and `Fred`, and that both `Joe` and `Bill` precede `Fred` in the file of records representing `Person`. Note that no precedence relationship exists between `Joe` and `Bill`. Therefore, the permutations of `Person` shown in Figures 5 (b) and (c) (differing only by the order of `Joe` and `Bill`) are the only

```
OrdCon [T]: trait
introduces
    none:                  → OrdCon [T]
    ord:   (T, T → Bool), OrdCon [T] → OrdCon [T]
    __ ≺ __ WITH __:   T, T, OrdCon [T] → Bool
    __ ≡ __ WITH __:   T, T, OrdCon [T] → Bool
    __ ⊒ __:   OrdCon [T], OrdCon [T] → Bool
    __ = __:   OrdCon [T], OrdCon [T] → Bool
asserts
 ∀ x,y,z: T, p: (T,T → Bool), o,o': OrdCon [T]

 % Ordering Axiom
 preord (p) ⟹
  p (x, y) ∧ ¬ p (y, x) ⇔ x ≺ y WITH ord (p, none)

 % Grouping Axiom
 preord (p) ⟹
  p (x, y) ∧ p (y, x)  ⇔  x ≡ y WITH ord (p, o)

 % Subgrouping Axiom
 preord (p) ⟹
  x ≺ y WITH ord (p, o)  ⇔
   (x ≺ y WITH ord (p, none)) ∨
   (x ≡ y WITH ord (p, o) ∧ x ≺ y WITH o)

 % Precedes or Same Group (''Not After'')
 x ⪯ y WITH o ⟺
  (x ≺ y WITH o) ∨
   ∀ z (z ≺ x WITH o ⇒ z ≺ y WITH o)

 % OrderConstraint Strength
 o ⊒ o' ⟺
  (x ≺ y WITH o ⇒ x ≺ y WITH o') ∧
  (x ≡ y WITH o ⇒ x ≡ y WITH o')

 % Equality Axiom
 o = o' ⟺ (o ⊒ o' ∧ o' ⊒ o)

 % Sorted Ordering Axiom
 sortcon (ord (p, o')) ⟺
  (sortord (p) ∧ (sortcon (o') ∨ o' = none))
```

Figure 6: An LSL Specification of `OrdCon`

possible ways that records from `Person` can be ordered according to this constraint.

The axioms of `OrdCon` translate an ordering constraint, `ord (`p`, `o`)`, into the precedence and grouping relationships that must hold between records in a file. The *Ordering Axiom* says that if the preorder predicate, p is true of a pair of elements, (x, y) but not true of (y, x), then x must precede y. In the example above, the ordering axiom determines that `Helen` must precede `Tom` because `leqAge` holds of `(Helen, Tom)` and not of `(Tom, Helen)`. The *Grouping Axiom* says that if p holds of both (x, y) and (y, x), then x and y are in the same group. Intuitively, records in the same group share precedence relationships with other records in the file. Namely, if x and y are in the same group, then any record that precedes (follows) x must precede (follow) y and vice-versa.

```
Index [T]: trait
introduces
 index:  File [T], OrdCons [T]   →  index [T]
 pind:   index [T]               →  Bool
 sind:   index [T]               →  Bool
 bkt:    index [T]               →  File [Int]
 ref:    File [T], Int           →  T
asserts
 ∀ x:  T, o, o':  OrdCons [T], i:  Int,
   F: File[T], F': File[Int]

   % Index Ordering Axiom
   ordcon (ord (p, o)) ⟺ sortord (p)

   % Primary Index Axiom
   ordcon (o) ∧ pind (index (F, o)) ⟹
     o = ordof (F)

   ordcon (o) ∧ pind (index (F, o)) ⟹
     ∀ o' (pind (index (F, o')) ⇒ (o' = o))

   % Secondary Index Axioms
   ordcon (o) ∧ sind (index (F, o)) ⟹
     x ∈ F ⇔ ∃ i ∈ bkt (index (F, o)) (ref_F (i) = x)

   ordcon (o) ∧ sind (index (F, o)) ⟹
     ordof (bkt (index (F, o))) = o ⊗ ref_F
```

Figure 7: An LSL Specification of `Index`

Suppose that the file for `Person` is specified as,

`File` (`Person`, `ord` (`equalDept`, `none`))

such that `equalDept` is true of (x, y) if x.`Dept` $= y$.`Dept`. In this case, the Ordering Axiom generates no precedence constraints (because `equalDept` is symmetric), but the Grouping Axiom establishes that all records in the file which have the same value for `Dept` are in the same group. Thus, this file representation for `Person` has two groups of records: one with `Dept = CS` and the other with `Dept = Eco`. This ordering constraint is satisfied by 24 different permutations of `Person` – two of which are shown in Figures 5d and e. All other permutations simply vary the order of `Helen` and `Fred`, and of `Tom`, `Joe` and `Bill`.

The subordering axiom establishes the ordering or grouping of records within a group. The two example ordering constraints discussed above both had '`none`' as their secondary ordering constraint (the second argument to `ord`) and therefore the Subgrouping Axiom generated no additional precedence information. But suppose that the file for `Person` is specified as,

`File` (`Person`, `ord` (`geqDept`, `ord` (`leqAge`, `none`)))

such that `geqDept` is true of (x, y) if x.`Dept` $\geq y$.`Dept` (according to a lexicographical ordering) and `leqAge` is defined as before. In this case, the Ordering Axiom establishes that `Helen` and `Fred` must precede `Tom`, `Joe` and `Bill`, but establishes no ordering between `Helen` and `Fred`, nor between `Tom`, `Joe` and `Bill`. The Grouping Axiom generates no additional information, but the Subgrouping Axiom establishes that groups of records that share the same value for `Dept` are ordered in ascending order on `Age`. Thus, this ordering constraint specifies the permutation of `Person` shown in Figure 5 (d) or one where `Helen` and `Fred` follow `Tom`, `Joe` and `Bill`, but no other.

Aside from the Ordering, Grouping and Subgrouping Axioms, the specification of Figure 6 also defines 1) the "Not After" relationship between records, which holds of (x, y) if x precedes or is in the same group as y, 2) the "Strength" relationship between ordering constraints, which holds of (o, o') if precedence (grouping) relationships specified by o are also specified by o' 3) ordering constraint equality, which holds if ordering constraints are of equal strength, and 4) the "Sorted Constraint" property (`sortcon`) of ordering constraints, which holds if the ordering constraint is defined in terms of a "Sort Order" predicate (See Figure 3). A sorted ordering constraint ensures that a file is indeed ordered and not just grouped.

3.2 Indexes

Figure 7 shows a formal specification of primary and secondary indexes. The constructor, `index`, takes a file, F, and a sorted ordering constraint (`ordcon`), o as inputs, and returns an index that can be scanned to retrieve the records of F in the order specified by o.

The index, $i =$ `index` (`F`, `o`) is a primary index over F if `pind` (i) holds and a secondary index over F if `sind` (i) holds. The first primary index axiom of Figure 7 specifies that the ordering retrievable by a primary index must be the physical ordering of the file itself. The second primary index axiom establishes that a file can have only one primary index.

The first secondary index axiom establishes a one-to-one relationship between records in a file with a secondary index, and integers contained in a *bucket* file for the index (`bkt` (i)). The bucket file of a secondary index contains integer references to the records in the indexed file. Thus, the contents of `bkt` (i) is a set of n integers (n is the number of records in F), such that for any integer, k, in this set, $\texttt{ref}_F(k)$ returns the record in F referenced by k (again, assuming that i is a secondary index for F).

The second secondary index axiom defines the ordering of integers in `bkt` (i), specifying an ordering constraint, $o \otimes \texttt{ref}_F$, constructed from `o`. Specifically, if there exist two records in F, x and y, such that $x \prec y$ `WITH` o and for integers v_1 and v_2:

$$\begin{aligned} \texttt{ref}_F\ (v_1) &= x, \text{ and} \\ \texttt{ref}_F\ (v_2) &= y, \end{aligned}$$

then $v_1 \prec v_2$ `WITH` $(o \otimes \texttt{ref}_F)$.

4 GPA: A Generic Plan Algebra

In this section, we describe our [G]eneric [P]lan [A]lgebra (GPA). Because of space limitations, we do not include the Larch axioms that define the operators of GPA. Instead, these axioms can be found in Appendix A.

The operators of GPA are not new – they were chosen after studying the source code of the Postgres query

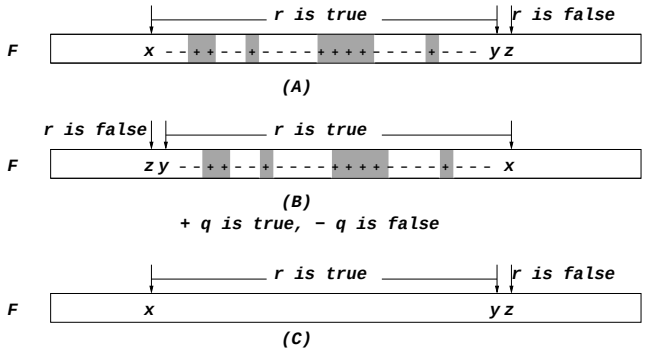

Figure 8: `Fscan`, `Bscan` and `Fmove` Illustrated

optimizer [34], and the large body of literature on access methods and processing (e.g., as summarized in [11] and standard database texts [24]). They are also not fixed – the algebra can easily be extended for new operators. Our contribution is the formal specification of the semantics of these operators, and the use of the formal specification to verify the correctness of *translation rules* that specify how to rewrite logical query expressions into plans, and of *refinement rules* that specify how to improve generated plans if certain conditions hold. In Section 5, we present some of the translation and refinement rules that have been formally verified with the Larch theorem prover [15].

4.1 Scanning: `fscan`, `bscan` and `fmove`

Given a pair of predicates, q (the "filter" predicate) and r (the "stop" predicate), and any record x belonging to a file, F, "$\texttt{fscan}_{q,r}((x,F))$" returns a file of records drawn from F that satisfy q. As shown in Figure 8a, this file of records will consist of records (shaded) starting at (and possibly including) x up to some record y such that:

1. all records in F from x to y inclusive, satisfy r, but the element *after* y (z) does not satisfy r,
2. the last record in F is y (and all records from x to the second to last record in F satisfy r), or
3. records y and x are the same (and either x doesn't satisfy r or x is the last record in F).

`Bscan` is similar to `fscan`, but moves backward rather than forward through a file. That is, for "filter" and "stop" predicates q and r, and record x belonging to file, F, "$\texttt{bscan}_{q,r}((x,F))$" returns a file of records that satisfy q, starting from (and possibly including) x and moving backwards. As shown in Figure 8b, "$\texttt{bscan}_{q,r}$ ((x, F))" returns a file that begins at some record, y, in F, such that:

1. all records in F from y to x inclusive, satisfy r, but the element *before* y (z) does not satisfy r,
2. the first record in F is y (and all records starting from the beginning of the file to x satisfy r), or
3. records y and x are the same (and either x doesn't satisfy r or x is the first record in F).

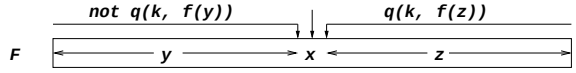

Figure 9: `Bsearch` and `Pisearch` illustrated

`Fmove` is like `fscan`, but it returns a record rather than a file. As shown in Figure 8c, for any "stop predicate", r, and record x belonging to file, F, "$\texttt{fmove}_r((x,F))$" returns the record in F, y, such that:

1. x precedes y, and all records in F from x to y satisfy the "stop predicate", r, but the record that immediately after y does not satisfy r,
2. y is the last record in the file (if all records from x to the second to last record in F satisfy r), or
3. y and x are the same record, (if x doesn't satisfy r or x is the last record in F).

In short, both `fmove` and `fscan` find the same record y that follows record x, but whereas `fscan` returns a file of records "between" x and y, `fmove` returns y itself.

4.2 Searching: `bsearch`, `pisearch`, `sisearch`

The search operators of GPA are binary search (`bsearch`), primary index search (`pisearch`) and secondary index search (`sisearch`). All of these operators search for some value, k in a file F, and return a record in F as a result.

Given a function, f, value, k, and a file, F, that is sorted in ascending ($q = \leq$) or descending ($q = \geq$) order on f, "$\texttt{bsearch}_{f,k}(F)$" searches for a record, x, in F such that $f(x) = k$. Such a record may not exist, so in general, as shown in Figure 9, a record x will be returned for which all of the following hold:

- for every record, y, that precedes x in F, $q(k, f(y))$ is false. For example, if q were $\leq$, f were `Age` and k were 21, then all records appearing before x would have age less than 21.
- for every record, z, that follows x in F, $q(k, f(z))$ holds. For example, if q were $\leq$, f were `Age` and k were 21, then x and all records following after x would have age greater than or equal to 21

The definitions above say that if there is no record for which `Age` is 21, the record returned is that whose value for `Age` is the smallest in F, but still larger than 21 (or analagously, whose value is largest in F, but still smaller than 21 if F is ordered in descending order). Note also that this definition specifies that if there is more than one record that has a value for `Age` of 21, the `bsearch` operator returns the first one appearing in F.

`Pisearch` has similar semantics to `bsearch`: given function, f, value, k, and primary index, i_p = $\texttt{index}(\texttt{F},\texttt{o})$, over some file, F that is ordered by an order constraint, o, specifying an ascending ($q = \leq$) or descending ($q = \geq$) ordering over attribute, f, "$\texttt{pisearch}_{\texttt{f,k}}(\texttt{i}_\texttt{p})$" searches for a record, x, in F such that $f(x) = k$. As with `bsearch`, the record x returned is that with the smallest value for f in F that is still

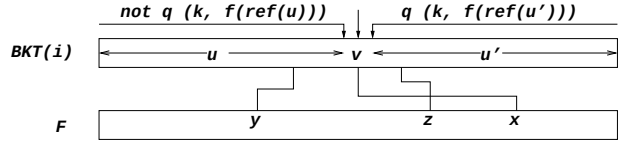

Figure 10: `Sisearch` illustrated

larger than k (if q is $\leq$) or that has the largest value for f in F that is still smaller than k (if q is $\geq$).

Given a function, f, value, k, and secondary index, i_s = `index(F,o)`, such that `o` specifies an ascending ($q = \leq$) or descending ($q = \geq$) ordering over attribute, f, "$\texttt{sisearch}_{\texttt{f,k}}(\texttt{i}_\texttt{s})$)" searches for a record, x, in F such that $f(x) = k$. The result is an integer, $v \in \texttt{bkt}(\texttt{i}_\texttt{s})$ such that $\texttt{ref}_\texttt{F}(\texttt{v}) = x$ where x has the smallest value for f in F that is still larger than k (if q is $\leq$) or that has the largest value for f in F that is still smaller than k (if q is $\geq$). Because o need not be an ordering of F, nothing can be said about the records that appear before or after x. However, the integers, u that precede v in $\texttt{bkt}(\texttt{i}_\texttt{s})$ reference records in F ($\texttt{ref}_\texttt{F}(\texttt{u})$) such that $q(k, f(\texttt{ref}_\texttt{F}(\texttt{u})))$ is false, and the integers, u' that follow v in $\texttt{bkt}(\texttt{i}_\texttt{s})$ all reference records in F ($\texttt{ref}_\texttt{F}(\texttt{u})$) such that $q(k, f(\texttt{ref}_\texttt{F}(\texttt{u})))$ is true.

4.3 Sorting: `emsort`, `pmsort`

GPA includes two sort operators: `emsort` is a generic external merge sort as defined in standard database texts [24]. `Pmsort` is a polyphase-merge sort discussed in [23] and implemented in Postgres [34].

`Emsort` is a generic, stable external merge sort. Given a "sort order" predicate (Figure 3), p, and a file, F, "$\texttt{emsort}_p(F)$" returns a file whose contents are the same as the contents of F, but whose ordering constraint is now based on p (i.e., for any two records, x and y in F, x will precede y in the result if $p(x, y)$ holds).

Because this sort is *stable*, the relative ordering of records x and y for which p establishes no order (i.e., $p(x, y) \wedge p(y, x)$) is the same in the result of the sort as it was in F. Therefore, if the file being sorted satisfies an ordering constraint, `ord (q, o)` and we use `emsort` to sort this file on predicate p, then the result satisfies ordering constraint, `ord (p, ord (q, o))`.

`Pmsort` is a polyphase external merge sort as defined in [23] and used in Postgres [34]. As with `emsort`, given a "sort order" predicate, p, and a file, F, "$\texttt{pmsort}_p(F)$" returns the contents of F sorted by p. Unlike `emsort`, this sort is not *stable*. Therefore, the ordering constraint of the result will always be `ord (p, none)`.

4.4 Joins: `nljoin`, `mjoin`

Given files, F_1 and F_2, and a predicate over pairs, p, $\texttt{nljoin}_p(F_1, F_2)$ returns a file containing paired records from F_1 and F_2 that satisfy p.

The ordering of records in the result of an `nljoin` operation is defined as follows. Let F_3 be the file that results from $\texttt{nljoin}_p(F_1, F_2)$. Let (x_1, y_1) and (x_2, y_2) be any two records in F_3 such that x_1 and x_2 are from F_1 and y_1 and y_2 are from F_2.

- If x_1 precedes x_2 in F_1, then (x_1, y_1) precedes (x_2, y_2) in F_3.
- If $x_1 = x_2$ and y_1 precedes y_2 in F_2, then (x_1, y_1) precedes (x_2, y_2) in F_3.
- If x_1 and x_2 are in the same group in F_1, then (x_1, y_1) and (x_2, y_2) are in the same group in F_3.
- If $x_1 = x_2$, and y_1 and y_2 are in the same group in F_2, then (x_1, y_1) and (x_2, y_2) are in the same group in F_3.

Given files, F_1 and F_2, that are ordered according to their values for attributes f and g respectively, $\texttt{mjoin}_{\texttt{f,g}}(\texttt{F}_1, \texttt{F}_2)$ returns a file containing pairs of records, (t, u) such that t is a record in F_1, u is a record in F_2, and $f(t) = g(u)$. The record orderings generated by `mjoin` are the same as those generated by `nljoin`.

4.5 Others: `Map`, `Grp`, `PickRep`

Given a file, F, and a function, f, $\texttt{map}_\texttt{f}(\texttt{F})$ returns the file of records resulting from applying f to every record in F in sequence. Given set-based definitions of files, this operator also removes duplicate records from the result.

Given $F_2 = \texttt{map}_f(F)$, the record orderings in F_2 are unknown, except when for any records x and y in F:

1. x precedes y in F, and
2. for all records in F, z, such that $f(x) = f(z)$, z precedes y in F.

In this case, $f(x)$ precedes $f(y)$ in F_2.

Given a file, F, and function, f, $\texttt{grp}_f(F)$ reorders the records in F so that all records, x and y, for which $f(x) = f(y)$, are put in the same group. Essentially, this operator changes the ordering constraint of a file without affecting its contents.

`Grp` is a stable operator: if record x precedes record y in F and $f(x) = f(y)$, then x will precede y in the file resulting from $\texttt{grp}_f(F)$. In other words, any ordering constraint associated with F becomes a secondary ordering constraint to that of the grouping in the result of the query.

`PickRep` is a nondeterministic operator that picks representative records out of a file. Given a file, F, and a function, f, $\texttt{pickrep}_f$ `(F)` will pick records from F such that for any two records, x and y, in the result, $f(x)$ and $f(y)$ are different, and such that for any record x in F, there exists a record, z, in the result such that $f(x) = f(z)$. Record orderings in F are preserved in $\texttt{pickrep}_f$ (F).

5 Translation

In this section, we demonstrate the expressibility of GPA by translating some common SQL queries into their plan equivalents. The queries translated are selection queries of the form, "`SELECT * FROM R WHERE p (R.Att, k)`" such that $p \in \{=, \leq, \geq, <, >\}$. The translations shown generate plans that perform binary search (`bsearch`),

primary index search (pisearch), and secondary index search (sisearch).[5]

The binary search and primary index search both demand that R be represented by a file with an ordering constraint that is a sorted constraint based on Att (i.e., by file (R, ord (q, o)) such that o is a sorted constraint and q is the sort order predicate defined by:

$$q(x, y) \Leftrightarrow x.\texttt{Att} \leq y.\texttt{Att}$$

(if the file ordered is in ascending order, and replacing "$\leq$" with "$\geq$" otherwise). We assume here that the file is ordered in ascending order, although it is straightforward to adjust the plans presented here for the case where it is ordered in descending order.[6] In describing primary index searches, we will also assume the existence of an index, i_p = index (R, ord (q, o)), such that pind (i_p) is true. The secondary index search does not require the file representing R to be sorted, but the index itself will be associated with the sorted constraint, ord (q, o), Here we will also assume that this ordering is ascending on Att, and that there exists an index, i_s = index (R, ord (q, o)), such that sind (i_s) is true.

The translations below use some non-query operators (next, prev, fst and lst) whose defining axioms are not given here due to space limitations, but which can be found in our technical report [35]. These operators are defined over files that are ordered according to sorted ordering constraints. Given such a file, F and record, r:

- fst (F) (lst(F)) which returns the first (last) record in F provided that F is not empty, and
- prev (r, F) (next(r, F)) which returns r if r is the first (last) record in F and the record that immediately precedes (follows) r otherwise.

Note that a file must be ordered according to a sorted ordering constraint to make these operations well-defined.

To simplify the presentation below, we use the notation:

- $\texttt{R}_{file}$ as shorthand for "file (R, o)",
- p* (for integer, p) as shorthand for, "$\texttt{ref}_{\texttt{R}_{file}}$(p)",
- $q(x)$ as shorthand for the predicate, "$q(x) \iff x.\texttt{Att} = \texttt{k}$", and
- $f(x)$ as shorthand for the function, $f(x) = x*$.

[5] The translation of these queries into a linear search is straightforward, always generating a plan of the form,

$$\texttt{fscan}_{q,\texttt{TRUE}}\,(\texttt{fst}\,(\texttt{file}\,(\texttt{R}, \texttt{none})))$$

such that $q(x) \Leftrightarrow p(x.\texttt{Att}, \texttt{k})$, which does a complete file scan of the file representation of R (file (R, none)) beginning at the first position in the file.

[6] For example, the plan generated for a query that selects tuples in R for which R.Att $\geq$ k assuming a file sorted in descending order, is the same plan generated for a query that selects tuples in R for which R.Att $\leq$ k assuming a file sorted in ascending order.

Query 1: SELECT * FROM R WHERE R.Att = k

Case 1: Att *a key*:

When Att is a key, the plans generated for this query simply perform a search for k, returning a file with a single record if k is found and empty otherwise. The secondary index search does the same thing, but must proceed through an extra level of indirection.

Binary Search:

```
1.  r ← bsearch Att,k (R_file)
2.  IF (r.Att = k) THEN return file ({r}, o)
    ELSE return empty
```

Primary Index Search:

```
1.  r ← pisearch Att,k (i_p)
2.  IF (r.Att = k) THEN return file ({r}, o)
    ELSE return empty
```

Secondary Index Search:

```
1.  p ← sisearch Att,k (i_s)
2.  IF (p*.Att = k) THEN return file ({p*}, o)
    ELSE return empty
```

Case 2: Att *not a key*:

If Att is not a key, the plans generated for this query must follow-up a search for k with a forward scan that stops when the first record is found for which Att $\neq$ k.

Binary Search:

```
1.  r ← bsearch Att,k (R_file)
2.  IF (r.Att = k) THEN
      return fscan q(x),TRUE (r, R_file)
    ELSE return empty
```

Primary Index Search:

```
1.  r ← pisearch Att,k (i_p)
2.  IF (r.Att = k) THEN
      return fscan q(x),TRUE (r, R_file)
    ELSE return empty
```

Secondary Index Search:

```
1.  p ← sisearch Att,k (i_s)
2.  IF (p*.Att = k) THEN
      return map f(x) (fscan q(x*),TRUE (p, bkt (I_s)))
```

Query 2: SELECT * FROM R WHERE R.Att $\geq$ k

This range query searches for k and then performs a forward scan to the end of the file. An exception is made if k is larger than the values of Att for all records in the file, in which case empty is returned.

Binary Search:

```
1.  r ← bsearch Att,k (R_file)
2.  IF (k > r.Att) THEN return empty
    ELSE return fscan TRUE,TRUE (r, R_file)
```

Primary Index Search:

```
1.  r ← pisearch Att,k (i_p)
2.  IF (k > r.Att) THEN return empty
    ELSE return fscan TRUE,TRUE (r, R_file)
```

Secondary Index Search:

```
1.  p ← sisearch Att,k (i_s)
2.  IF (k > p*.Att) THEN return empty
    ELSE return map f(x) (fscan TRUE,TRUE (p, bkt (I_s)))
```

Query 3: SELECT * FROM R WHERE R.Att < k

This range query searches for k and then performs a backward scan starting from the position previous to the position returned by the search, and proceeding to the beginning of the file. An exception is made if k is less than or equal than the values of Att for all records in the file, in which case empty is returned.

Binary Search:

```
1.  r ← bsearch_{Att,k} (R_file)
2.  IF (r = fst (R_file)) THEN return empty
    ELSIF (k ≤ r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE return bscan_{TRUE,TRUE} (r, R_file)
```

Primary Index Search:

```
1.  r ← pisearch_{Att,k} (i_p)
2.  IF (r = fst (R_file)) THEN return empty
    ELSIF (k ≤ r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE return bscan_{TRUE,TRUE} (r, R_file)
```

Secondary Index Search:

```
1.  p ← sisearch_{Att,k} (i_s)
2.  IF (p = fst (bkt (i_s))) THEN return empty
    ELSIF (k ≤ p*.Att) THEN
      return bscan_{TRUE,TRUE} (prev (p, bkt (I_s)))
    ELSE
      return map_{f(x)} (bscan_{TRUE,TRUE} (p, bkt (I_s)))
```

Query 4: SELECT * FROM R WHERE R.Att ≤ k

Case 1: Att *a key*:

This range query searches for k and then performs a backward scan. An exception is made if k is less than than the values of Att for all records in the file, in which case empty is returned.

Binary Search:

```
1.  r ← bsearch_{Att,k} (R_file)
2.  IF (r = fst (R_file)) ∧ (k < r.Att) THEN
      return empty
    ELSIF (k ≥ r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE return bscan_{TRUE,TRUE} (r, R_file)
```

Primary Index Search:

```
1.  r ← pisearch_{Att,k} (i_p)
2.  IF (r = fst (R_file) ∧ k < r.Att) THEN
      return empty
    ELSIF (k ≥ r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE return bscan_{TRUE,TRUE} (r, R_file)
```

Secondary Index Search:

```
1.  p ← sisearch_{Att,k} (i_s)
2.  IF (p = fst (bkt (i_s)) ∧ k < p*.Att) THEN
      return empty
    ELSIF (k ≥ p*.Att) THEN return
      map_{f(x)} (bscan_{TRUE,TRUE} (prev (p, bkt (I_s))))
    ELSE return
      map_{f(x)} (bscan_{TRUE,TRUE} (p, bkt (I_s)))
```

Case 2: Att *not a key*:

If Att is not a key, this range query must follow-up a search for k with a move forward to the last record for which Att = k and then follow with a backward scan.

Binary Search:

```
1.  r ← bsearch_{Att,k} (R_file)
2.  IF (r = fst (R_file) ∧ k < r.Att) THEN
      return empty
    ELSIF (k < r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE
      return bscan_{TRUE,TRUE} (fmove_{q(x)} (r, R_file))
```

Primary Index Search:

```
1.  r ← pisearch_{Att,k} (i_p)
2.  IF (r = fst (R_file) ∧ k < r.Att) THEN
      return empty
    ELSIF (k < r.Att) THEN
      return bscan_{TRUE,TRUE} (prev (r, R_file))
    ELSE return bscan_{TRUE,TRUE} (fmove_{q(x)} (r, R_file))
```

Secondary Index Search:

```
1.  p ← sisearch_{Att,k} (i_p)
2.  IF (p = fst (bkt (i_s)) ∧ k < p*.Att) THEN
      return empty
    ELSIF (k < p*.Att) THEN
      return bscan_{TRUE,TRUE} (prev (p, bkt (I_s)))
    ELSE return map_{f(x)} (bscan_{TRUE,TRUE} (e))
    s.t.  e = fmove_{q(x*)} (p, bkt (I_s))
```

Query 5: SELECT * FROM R WHERE R.Att > k

Case 1: Att *a key*:

If Att is a key, this range query must follow-up a search for k by moving foward one position before proceeding with a forward scan.

Binary Search:

```
1.  r ← bsearch_{Att,k} (R_file)
2.  IF (r = lst (R_file)) THEN return empty
    ELSIF (k = r.Att) THEN
      return fscan_{TRUE,TRUE} (next (r, R_file))
    ELSE return fscan_{TRUE,TRUE} (r, R_file)
```

Primary Index Search:

```
1.  r ← pisearch_{Att,k} (i_p)
2.  IF (r = lst (R_file)) THEN return empty
    ELSIF (k = r.Att) THEN
      return fscan_{TRUE,TRUE} (next (r, R_file))
    ELSE return fscan_{TRUE,TRUE} (r, R_file)
```

Secondary Index Search:

```
1.  p ← sisearch_{Att,k} (i_s)
2.  IF (p = lst (bkt (i_s))) THEN return empty
    ELSIF (k = p*.Att) THEN
      return
        map_{f(x)} (fscan_{TRUE,TRUE} (next (p, bkt (I_s))))
    ELSE return map_{f(x)} (fscan_{TRUE,TRUE} (p, bkt (I_s)))
```

Case 2: Att *not a key*:

Because Att is not a key, this range query must follow-up a search for k by moving foward to one record past the last record for which Att = k before proceeding with a forward scan.

Binary Search:

```
1.  r ← bsearch Att,k (R_file)
2.  IF (r = lst (R_file)) THEN return empty
    ELSIF (k = r.Att) THEN
      return fscan TRUE,TRUE (fmove q(x*) (r, R_file))
    ELSE return fscan TRUE,TRUE (fmove q(x*) (r, R_file))
```

Primary Index Search:

```
1.  r ← pisearch Att,k (i_p)
2.  IF (r = lst (R_file)) THEN return empty
    ELSIF (k = r.Att) THEN
      return fscan TRUE,TRUE (fmove q(x*) (r, R_file))
    ELSE
      return fscan TRUE,TRUE (fmove q(x*) (r, R_file))
```

Secondary Index Search:

```
1.  p ← sisearch Att,k (i_s)
2.  IF (p = lst (bkt (i_s))) THEN return empty
    ELSIF (k = p*.Att) THEN
      return map f(x) (fscan TRUE,TRUE (e))
    ELSE
      return map f(x) (fscan TRUE,TRUE (e))
      s.t.  e = fmove q(x*) ((p, bkt (I_s)))
```

6 Verification

In this section, we present by way of example, some of the translation and refinement rules over GPA operators that have been formally verified with the Larch theorem prover [15].

6.1 Translation Rules

A translation rule that translates query Q to plan P can be verified by showing that for any file, F representing some relation, R, that $Q(R) = \texttt{setof}\ (P(F))$.

Three example translation rules that have been verified in Larch are shown below. The first shows that a relational join query of sets A and B can be translated into a nested loop join of any file representations of A and B, regardless of the ordering of records in these files. The second translation rule says that any equijoin of sets A and B can be translated into a merge join of file representations of A and B that have been sorted on f and g respectively (any version of this rule that substitutes `emsort` for either or both occurences of `pmsort` is also correct). The final translation rule shows that a selection query can be translated into a filter scan.

(1) $A \bowtie_p B =$
$\texttt{setof}(\texttt{nljoin}_p(\texttt{file}(A, o_1), \texttt{file}(B, o_2)))$

(2) $A \bowtie_{f=g} B =$
$\texttt{setof}\ (\texttt{mjoin}_{f,g}(\texttt{pmsort}_{p \oplus (f \times f)}(\texttt{file}(A, o_1)),$
$\texttt{pmsort}_{p \oplus (g \times g)}(\texttt{file}(B, o_2))))$

(3) $\sigma_p (A) =$
$\texttt{setof}(\texttt{fscan}_{p,\ \texttt{TRUE}}(\texttt{fst}(\texttt{file}(A, o)), \texttt{file}(A, o)))$

6.2 Refinement Rules

A refinement rule that refines a plan P_1 into a plan P_2 can be verified by showing that for any file, F, representing a relation, that all record ordering relationships that result from $P_1\ (F)$ are preserved (perhaps with more imposed) by $P_2\ (F)$. We express this formally by saying that P_1 can be refined into P_2 provided that for any file, F:

$\texttt{setof}\ (P_1\ (F)) = \texttt{setof}\ (P_2\ (F))$, and
$\texttt{ordof}\ (P_1\ (F)) \sqsupseteq \texttt{ordof}\ (P_2\ (F))$.

Three example refinement rules that have been verified in Larch are shown below. These rules all eliminate unnecessary sort operations from plans. The first two rules say that applying a polyphase merge sort to the result of either another polyphase merge sort, or a stable external merge sort, eliminates the ordering of records generated by the initial sort. Because the effects of the initial sort are undone by the second, the initial sort need not be done at all.

The third refinement rule indicates that under certain conditions, a plan that performs two consecutive external merge sorts can be replaced by a plan that performs just one of them (the "outer" sort). The conditions require that there be a functional dependency from the function, f upon which the outer sort is based, to the function, g upon which the inner sort is based ($fd\ (f,\ g)$). Intuitively, this because performing a stable sort as an outer sort will preserve the ordering of the inner sort within groups produced by the outer sort (i.e., within those records whose values for f are the same). To achieve the same result without first sorting on g, it must be the case that the sorting on f automatically produces a sorting on g for records that have the same value for f. This is trivially the case if there is a functional dependency from f to g, as records that have the same value for f will also have the same value for g.[7]

(4) $\texttt{sortord}\ (p) \wedge \texttt{sortord}\ (q) \implies$
$\texttt{pmsort}_p\ (\texttt{pmsort}_q\ (F)) = \texttt{pmsort}_p\ (F)$

(5) $\texttt{sortord}\ (p) \wedge \texttt{sortord}\ (q) \implies$
$\texttt{pmsort}_p\ (\texttt{emsort}_q\ (F)) = \texttt{pmsort}_p\ (F)$

(6) $\texttt{sortord}\ (p \oplus (f \times f)) \wedge$
$\texttt{sortord}\ (q \oplus (g \times g)) \wedge fd\ (f,\ g) \implies$
$\texttt{emsort}_{p \oplus (f \times f)}\ (\texttt{emsort}_{q \oplus (g \times g)}\ (F)) =$
$\texttt{emsort}_{p \oplus (f \times f)}\ (F)$

7 Related Work

While many physical algebras have been proposed in the past (e.g., [26, 33]), as far as we know none have been defined with a formal specification. We make no claims of uniqueness of our operators. In fact, our choice of operators was inspired by studying the literature and the source code of an optimizer [34]. But our goal of being able to verify the correctness of plan generating rewrites performed by an optimizer is unique amongst plan algebra proposals.

Our notion of ordering constraints is inspired by the ordering optimizations of [31] and [32]. Both systems described use an attribute list to represent the ordering properties of a relation. DB2 [31] names "interesting orders" that must be satisfied by inputs to QGM-based operators. The goal here is to push down "interesting

[7] For these rules, the KOLA predicate, "$p \oplus (f \times f)$", is used to denote a predicate that is true of the pair (x, y) *iff* $p(f(x), f(y))$.

orders" and perform "early sorts" to reduce the number of sort operations performed. To express grouping, additional information is added, indicating which attributes can be permuted and which can be ordered either in ascending or descending order. [32] uses transformation rules to generate all alternative plans. To solve the problem of expressing grouping, rules are classified into six categories according to whether they preserve list equivalence, multiset equivalence set equivalence etc.

8 Conclusions

As extremely large and complex database components, query optimizers are brittle and error-prone. Our goal is to build a query optimizer generator that generates optimizers that can be formally verified as correct.

This paper presents work that moves us closer to our goal. We have presented a generic plan algebra and given it a formal specification. We have also formally specified the files of records that plan operators act on and return, ordering constraints that specify the ordering or grouping of records in files, as well as record positions and indexes. We have demonstrated the viability of this algebra by presenting a family of SQL queries and GPA plans that implement them. Finally, we have shown some of the translation and refinement rules that can be verified with an automated theorem prover assuming a GPA-based representation of plans. This brings us closer to being able to formally verify the safety-critical component of generated query planners.

References

[1] Ludger Becker and Ralf Hartmut Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, June 1992.

[2] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenberg. The EXODUS extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1990.

[3] Mitch Cherniack. *Building Query Optimizers with Combinators.* PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910, December 1998.

[4] Mitch Cherniack and Stan Zdonik. Changing the rules: Transformations for rule-based optimizers. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 61–72, Seattle, WA, June 1998.

[5] Mitch Cherniack and Stan Zdonik. Inferring function semantics to optimize queries. In *Proc. 24th Int'l Conference on Very Large Data Bases*, New York, NY, August 1998.

[6] Mitch Cherniack and Stanley B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Montréal, Québec, Canada, June 1996.

[7] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, December 1996.

[8] Beatrice Finance and Georges Gardarin. A rule-based query optimizer with multiple search strategies. *Data and Knowledge Engineering*, 13:1–29, 1994.

[9] Hubert Garavel and Laurent Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 1996. Special issue on Industrially Relevant Applications of Formal Analysis Techniques.

[10] Bill Gates, June 1998. Keynote Address at SIGMOD '98.

[11] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[12] Goetz Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3):19–29, September 1995.

[13] Goetz Graefe and Willam J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Vienna, Austria, April 1993. IEEE.

[14] Jim Gray. What next? a few remaining it problems, 1998. ACM Turing Award lecture.

[15] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specifications.* Springer-Verlag, 1992.

[16] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme". *Lisp and Symbolic Computation*, 8(1):1–28, 1995.

[17] Laura M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 377–388, 1989.

[18] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[19] Paula Hawthorne. Unsafe at any speed: database software development issues in the 20th century, October 1998. Presentation at NSF Workshop on Industrial/Academic Cooperation in Database Systems.

[20] I. Houston and S. King. CICS project report: Experiences and results from using Z. In *Proceedings of VDM '91: Formal Development Methods*, volume 551 of *LNCS*. Springer-Verlag, 1991.

[21] W. Kiessling. SQL-like and QUEL-like correlation queries with aggregates revisited. UCB/ERL Memo 84/75, Electronics Research Laboratory, Univ. California, Berkeley, 1984.

[22] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[23] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[24] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* Computer Science Series. McGraw-Hill, 1986.

[25] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[26] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *In Proc. ACM SIGMOD, 1988*, pages 18–27, 1988.

[27] Steven P. Miller, David A. Greve, and Mandayam K. Srivas. Formal verification of the AAMP5 and AAMP-FV microcode. In *Proceedings of the Third AMAST Workshop on Real-Time Systems*, Salt Lake City, Utah, March 1996.

[28] J. Ostroff. Formal methods for the specification and design of real-time safety-critical systems. *Journal of Systems and Software*, 18(1):33–60, April 1992.

[29] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 39–48, San Diego, CA, June 1992.

[30] Edward Sciore and John Sieg Jr. A modular query optimizer generator. In *Proceedings of the 6th International Conference on Data Engineering*, pages 146–153, Los Angeles, USA, 1990.

[31] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *In Proc. ACM SIGMOD 1996*, pages 57–67, 1996.

[32] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Query plans for conventional and temporal queries involving duplicates and ordering. In *ICDE*, pages 547–558, 2000.

[33] Michael Stillger, Guy Lohman, Volker Markl, and Makhtar Kandil. LEO - DB2's LEarning Optimizaer. In *VLDB 2001*, pages 19–28, 2001.

[34] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[35] Xiaoyu Wang and Mitch Cherniack. Gpa: A plan algebra for provably correct query planners. Technical report, Brandeis University Department of Computer Science, February 2002.

A A Formal Semantics of GPA

The following is a set of Larch *axioms* defining the operators of GPA.

A.1 Scanning Operators

A.1.1 Fscan: Forward Scan

$\forall$ x, y, z: T, p, r: (T $\rightarrow$ Bool), F: File [T]

ordof (fscan$_{p,\,r}$(x, F)) = ordof (F)

y $\in$ setof (fscan$_{p,\,r}$(x, F)) $\iff$
 (x $\preceq$ y WITH ordof (F) $\wedge$ p (y) $\wedge$
 $\forall$ z (z $\in$ setof (F) $\wedge$ x $\preceq$ z WITH ordof (F) $\wedge$
  z $\preceq$ y WITH ordof (F) $\Rightarrow$ r (z)))

A.1.2 BScan: Backwords Scan

$\forall$ x, y, z: T, p, r: (T $\rightarrow$ Bool), F: File [T]

x $\prec$ y WITH ordof (bscan$_{p,\,r}$(x, F)) $\iff$
 y $\prec$ x WITH ordof (F)

y $\in$ setof (bscan$_{p,\,r}$(x, F)) $\iff$
 (y $\preceq$ x WITH ordof (F) $\wedge$ p (y) $\wedge$
 $\forall$ z (z $\in$ setof (F) $\wedge$ z $\preceq$ x WITH ordof (F) $\wedge$
  y $\preceq$ z WITH ordof (F) $\Rightarrow$ r (z)))

A.1.3 FMove: Forward Move

$\forall$ x, y, z, k: T, p:(T $\rightarrow$ Bool), F: File [T]

y = fmove$_p$(x, F) $\iff$
 (y $\in$ setof (F) $\wedge$ x $\preceq$ y WITH ordof (F) $\wedge$
 $\forall$ z (z $\in$ setof (F) $\Rightarrow$
  (z $\preceq$ y WITH ordof (F) $\Rightarrow$ (p (z) $\vee$ y = x)) $\vee$
  (y $\prec$ z WITH ordof (F) $\Rightarrow$
   $\exists$ k (k $\in$ setof (F) $\wedge$ y $\prec$ k WITH ordof (F) $\wedge$
   k $\preceq$ z WITH ordof (F) $\wedge$ $\neg$ p (k)))))

A.2 Search Operators

A.2.1 BSearch: Binary Search

$\forall$ x, y: T, f: (T $\rightarrow$ U), F: File [T], k: U, p: (U, U $\rightarrow$ Bool)

ord (p $\oplus$ (f $\times$ f), none) $\sqsupseteq$ ordof (F) $\Longrightarrow$
 x = bsearch$_{f,\,k}$(F) $\Leftrightarrow$
  $\forall$ y (y $\in$ setof (F) $\Rightarrow$
   (y $\prec$ x WITH ordof (F) $\Rightarrow$ p (f (y), k)) $\wedge$
   (x $\prec$ y WITH ordof (F) $\Rightarrow$ $\neg$ p (f (y), k)))

A.2.2 Pisearch: Primary Index Search

$\forall$ F: File [T], p: (U, U $\rightarrow$ Bool), x, y: T, k: U,
 f: (T $\rightarrow$ U), o: OrdCons [T]

pind (index (F, ord (p $\oplus$ (f $\times$ f), o))) $\Longrightarrow$
 x = pisearch$_{f,\,k}$(index (F, ord (p $\oplus$ (f $\times$ f), o))) $\Leftrightarrow$
  $\forall$ y (y $\in$ setof (F) $\Rightarrow$
   (y $\prec$ x WITH ordof (F) $\Rightarrow$ p (f (y), k)) $\wedge$
   (x $\prec$ y WITH ordof (F) $\Rightarrow$ $\neg$ p (f (y), k)))

A.2.3 Sisearch: Secondary Index Search

$\forall$ F: File [T], f:(T $\rightarrow$ U), p:(U, U $\rightarrow$ Bool), k:U,
 i, j: Int, o: OrdCons [T], I: File [Int]

sind (index (F, ord (p $\oplus$ (f $\times$ f), o))) $\Longrightarrow$
 (i = sisearch$_{f,\,k}$(index (F, ord (p $\oplus$ (f $\times$ f), o))) $\wedge$
  I = bkt (index (F, ord (p $\oplus$ (f $\times$ f), o))) $\Leftrightarrow$
  $\forall$ j (j $\in$ I $\Rightarrow$
   (j $\prec$ i WITH ordof (I) $\Rightarrow$ p (f (ref$_F$(j)), k)) $\wedge$
   (i $\prec$ j WITH ordof (I) $\Rightarrow$ $\neg$ p (f (ref$_F$(j)), k))))

A.3 Sorting Operators

A.3.1 Emsort: External Merge Sort

$\forall$ p: (T, T $\rightarrow$ Bool), F: File [T]

sortord (p) $\Longrightarrow$
 emsort$_p$(F) = file (setof (F), ord (p, ordof (F)))

A.3.2 Pmsort: Polyphase Merge Sort

$\forall$ p (T, T $\rightarrow$ Bool), F: file [T]

sortord (p) $\Longrightarrow$
 pmsort$_p$(F) = file (setof (F), ord (p, none))

A.4 Join Operators

A.4.1 NLJoin: Nested Loop Join

$\forall$ x$_1$, x$_2$: T$_1$, y$_1$, y$_2$: T$_2$, F$_1$: File [T$_1$], F$_2$: File [T$_2$],
 o: OrdCons [(T$_1$, T$_2$)], p:(T$_1$, T$_2$ $\rightarrow$ Bool)

setof (nljoin$_p$(F$_1$, F$_2$)) = $\bowtie_p$ (setof (F$_1$), setof (F$_2$))

o = ordof (nljoin$_p$(F$_1$, F$_2$)) $\iff$
 (($x_1 \prec x_2$ WITH ordof (F$_1$) $\Rightarrow$ (x$_1$, y$_1$) $\prec$ (x$_2$, y$_2$) WITH o) $\wedge$
 (y$_1$ $\prec$ y$_2$ WITH ordof (F$_2$) $\Rightarrow$ (x$_1$, y$_1$) $\prec$ (x$_1$, y$_2$) WITH o) $\wedge$
 (x$_1$ $\equiv$ x$_2$ WITH ordof (F$_1$) $\Rightarrow$ (x$_1$, y$_1$) $\equiv$ (x$_2$, y$_2$) WITH o) $\wedge$
 (y$_1$ $\equiv$ y$_2$ WITH ordof (F$_2$) $\Rightarrow$ (x$_1$, y$_1$) $\equiv$ (x$_1$, y$_2$) WITH o))

A.4.2 MJoin: Merge Join

$\forall$ x$_1$, x$_2$: T$_1$, y$_1$, y$_2$: T$_2$, F$_1$: File [T$_1$], F$_2$: File [T$_2$],
 o: OrdCons [(T$_1$, T$_2$)], f: (T$_1$ $\rightarrow$ U), g:(T$_2$ $\rightarrow$ U)

setof (mjoin$_{f,\,g}$(F$_1$, F$_2$)) = $\bowtie_{=\oplus(f\times g)}$ (setof (F$_1$), setof (F$_2$))

o = ordof (mjoin$_{f,g}$(F$_1$, F$_2$)) $\iff$
 (($x_1 \prec x_2$ WITH ordof (F$_1$) $\Rightarrow$ (x$_1$, y$_1$) $\prec$ (x$_2$, y$_2$) WITH o) $\wedge$
 (y$_1$ $\prec$ y$_2$ WITH ordof (F$_2$) $\Rightarrow$ (x$_1$, y$_1$) $\prec$ (x$_1$, y$_2$) WITH o) $\wedge$
 (x$_1$ $\equiv$ x$_2$ WITH ordof (F$_1$) $\Rightarrow$ (x$_1$, y$_1$) $\equiv$ (x$_2$, y$_2$) WITH o) $\wedge$
 (y$_1$ $\equiv$ y$_2$ WITH ordof (F$_2$) $\Rightarrow$ (x$_1$, y$_1$) $\equiv$ (x$_1$, y$_2$) WITH o))

A.5 Other Operators

A.5.1 Map: Map

$\forall$ F: File [T], f: (T $\rightarrow$ U), x, y, z: T

x $\in$ setof (F) $\Longrightarrow$ f(x) $\in$ setof (map_{f}(F))

x $\in$ setof (F) $\wedge$ y $\in$ setof (F) $\wedge$ x $\prec$ y WITH ordof (F) $\wedge$
 $\forall$ z (z $\in$ setof (F) $\wedge$ f(x) = f(z) $\Rightarrow$ z $\prec$ y WITH ordof (F))
$\Longrightarrow$ f(x) $\prec$ f(y) WITH ordof (map_{f}(F))

A.5.2 Grp: Group

$\forall$ f: (T $\rightarrow$ U), F, F': File [T], x, y: T

setof (grp_{f}(F)) = setof (F)

F' = grp_{f}(F) $\Longrightarrow$
 (f(x) = f(y) $\wedge$ x $\prec$ y WITH ordof (F) $\Rightarrow$
  x $\prec$ y WITH ordof (F')) $\wedge$
 (f(x) = f(y) $\Rightarrow$ x $\equiv$ y WITH ordof (F'))

A.5.3 Pickrep: Pick Representatives

$\forall$ x, y: T, F, F': File [T], g: (T$\rightarrow$U)

ordof ($\text{pickrep}_{\text{f}}$(F)) = ordof (F)

F' = $\text{pickpep}_{\text{f}}$ (F) $\Longleftrightarrow$
 (x $\in$ setof (F') $\Rightarrow$ x $\in$ setof (F)) $\wedge$
 (x $\in$ setof (F) $\Rightarrow$ $\exists$ y (y $\in$ setof (F') $\wedge$ g (x) = g (y))) $\wedge$
 (x $\in$ setof (F') $\wedge$ y $\in$ setof (F') $\Rightarrow$
  (x = y) $\vee$ (g (x) = g (y)))