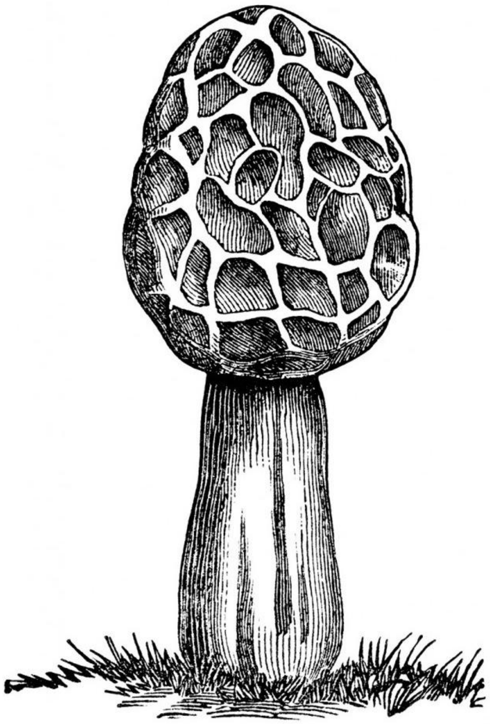
Is there a perfect data-parallel programming language?

Experiments with Morel and Apache Calcite

Julian Hyde @julianhyde (Looker/Google)

March 7th, 2022

ADCF FORUM



Agenda

Programming language types (and how they might be unified)

Data parallel
(e.g. Flume, Spark)

Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

Agenda

1. Data parallel
2. Functional
3. Functional + Relational
4. Functional + Data parallel
5. Functional + Deductive

Data parallel
(e.g. Flume, Spark)

Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

Agenda

1. **Data parallel**
2. Functional
3. Functional + Relational
4. Functional + Data parallel
5. Functional + Deductive

Data parallel
(e.g. Flume, Spark)

Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

Agenda

1. Data parallel
2. **Functional**
3. Functional + Relational
4. Functional + Data parallel
5. Functional + Deductive

Data parallel
(e.g. Flume, Spark)

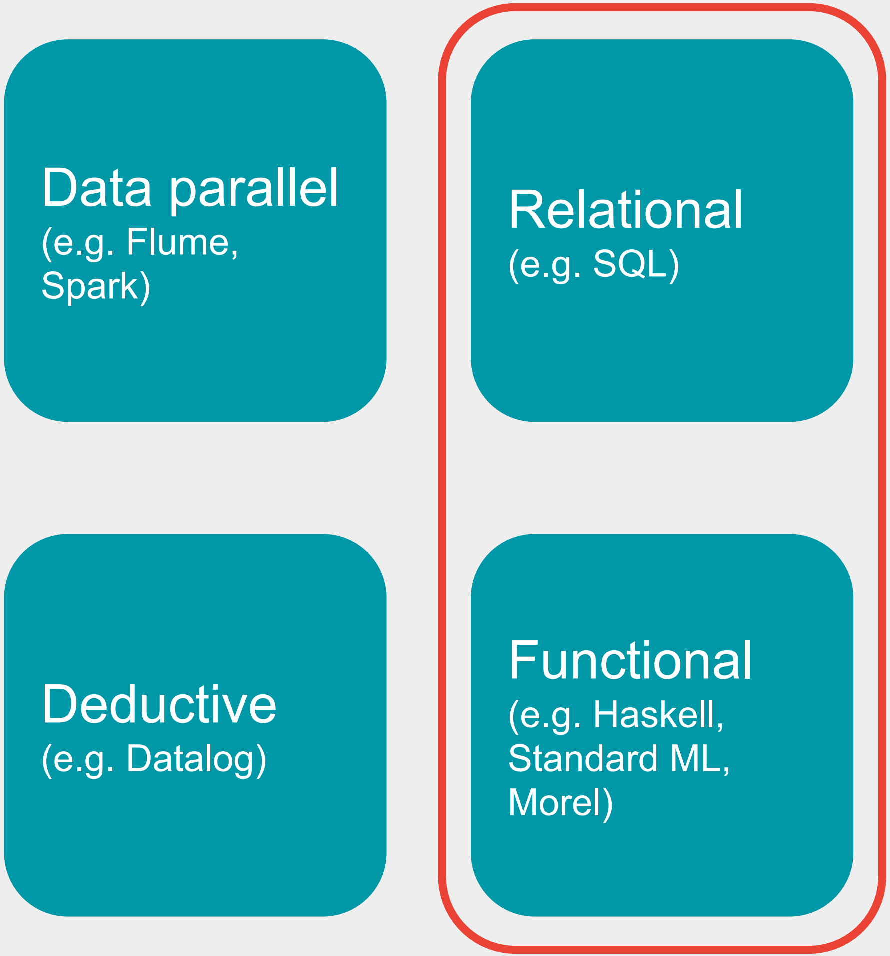
Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

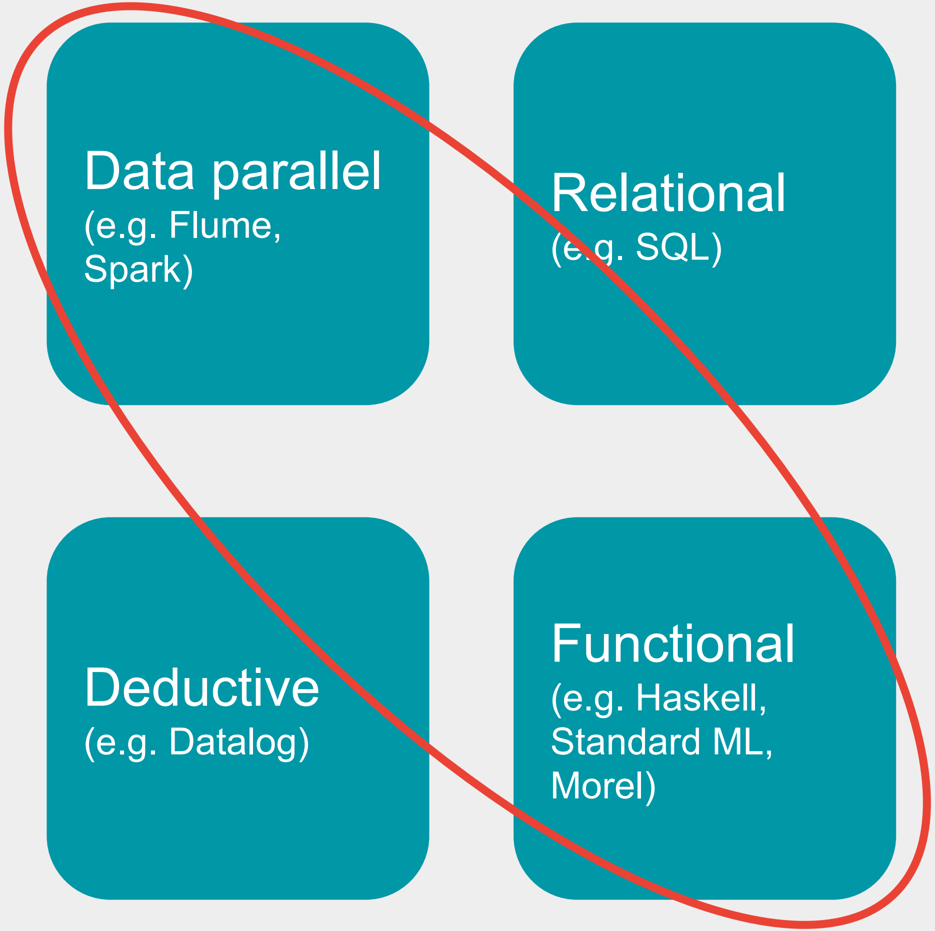
Agenda

1. Data parallel
2. Functional
3. **Functional + Relational**
4. Functional + Data parallel
5. Functional + Deductive



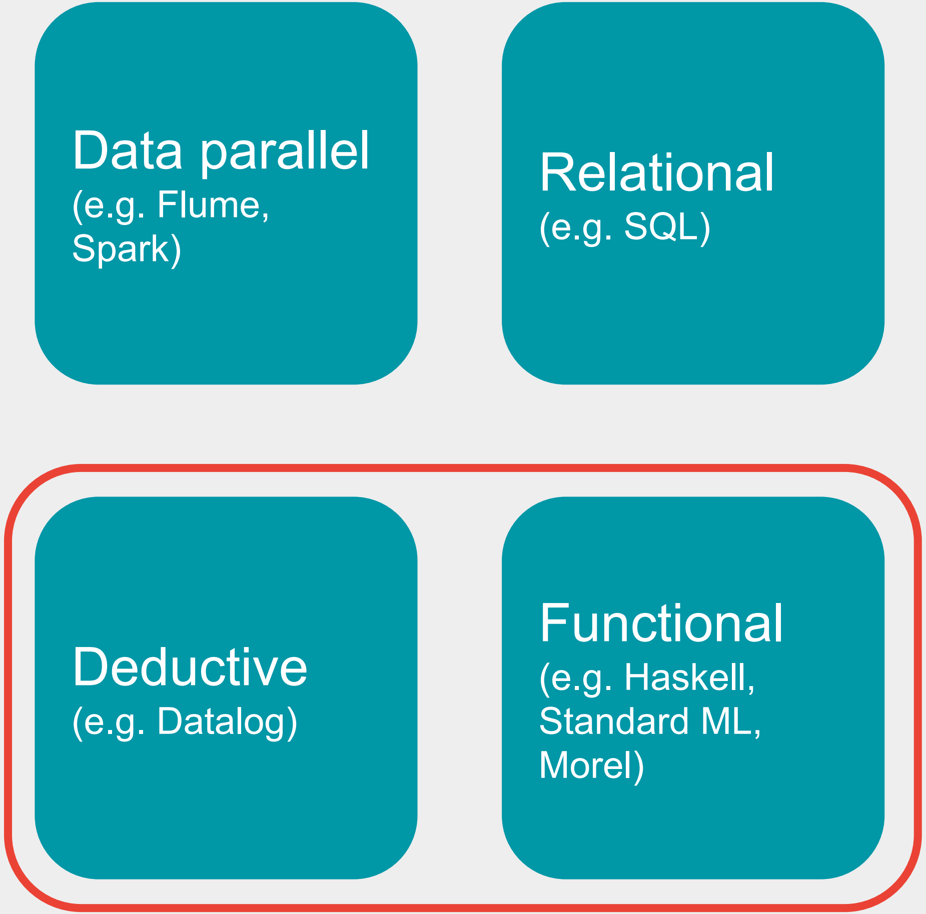
Agenda

1. Data parallel
2. Functional
3. Functional + Relational
4. **Functional + Data parallel**
5. Functional + Deductive



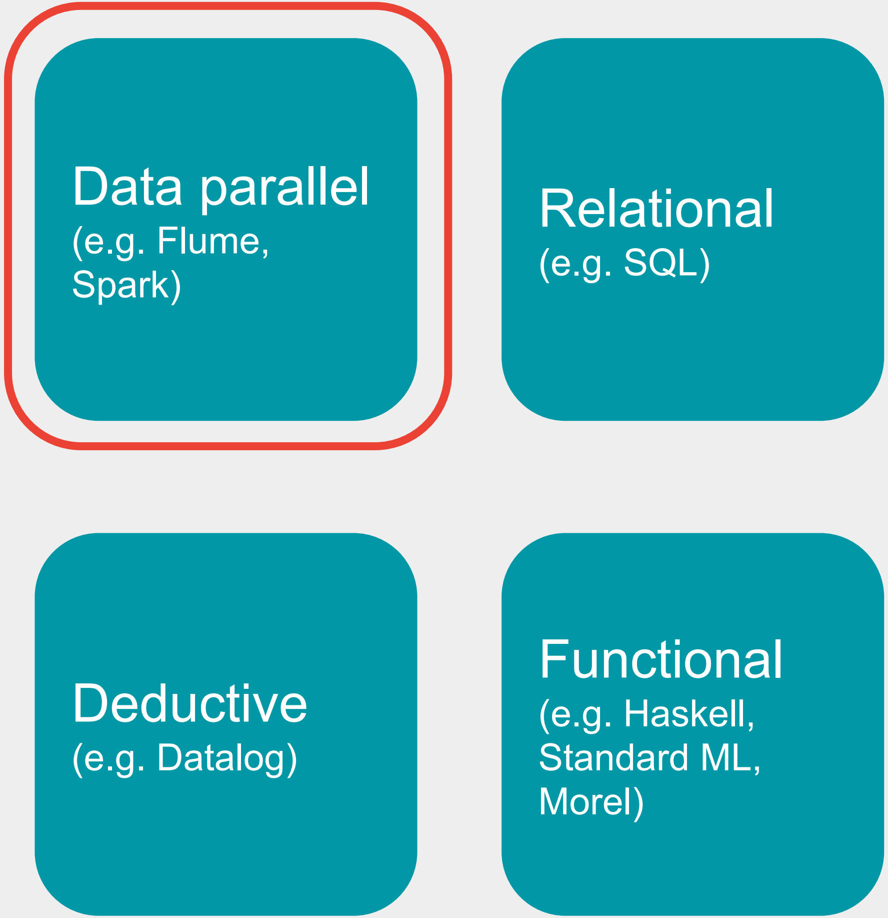
Agenda

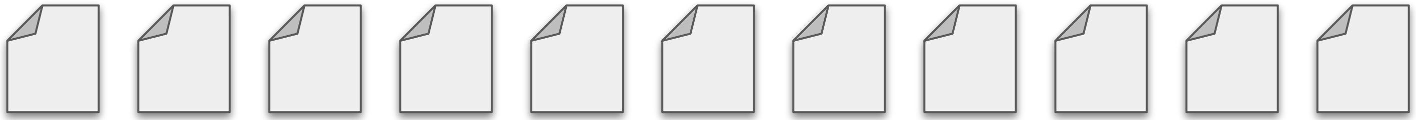
1. Data parallel
2. Functional
3. Functional + Relational
4. Functional + Data parallel
5. **Functional + Deductive**



1. Data parallel

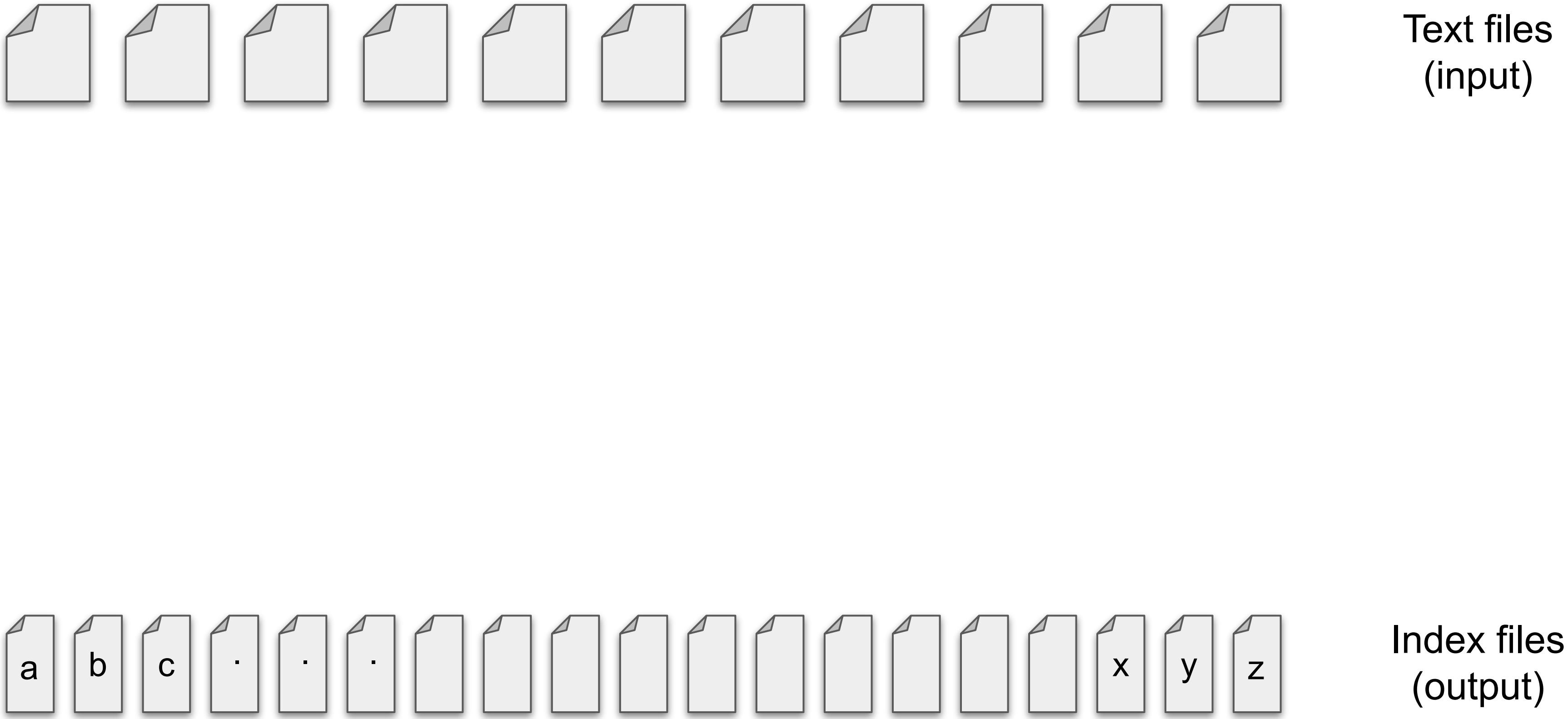
Explained via two data parallel problems (text indexing and WordCount)



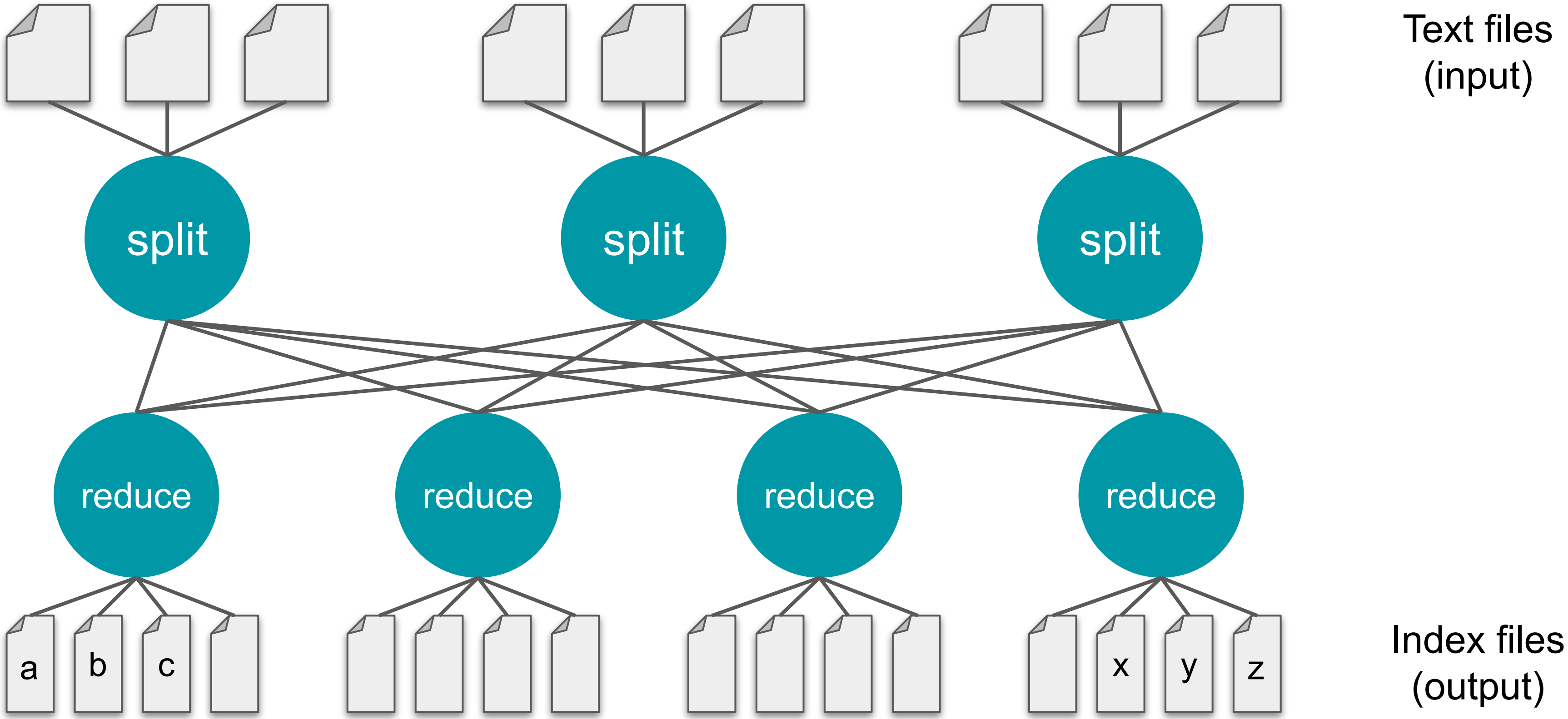


Text files
(input)

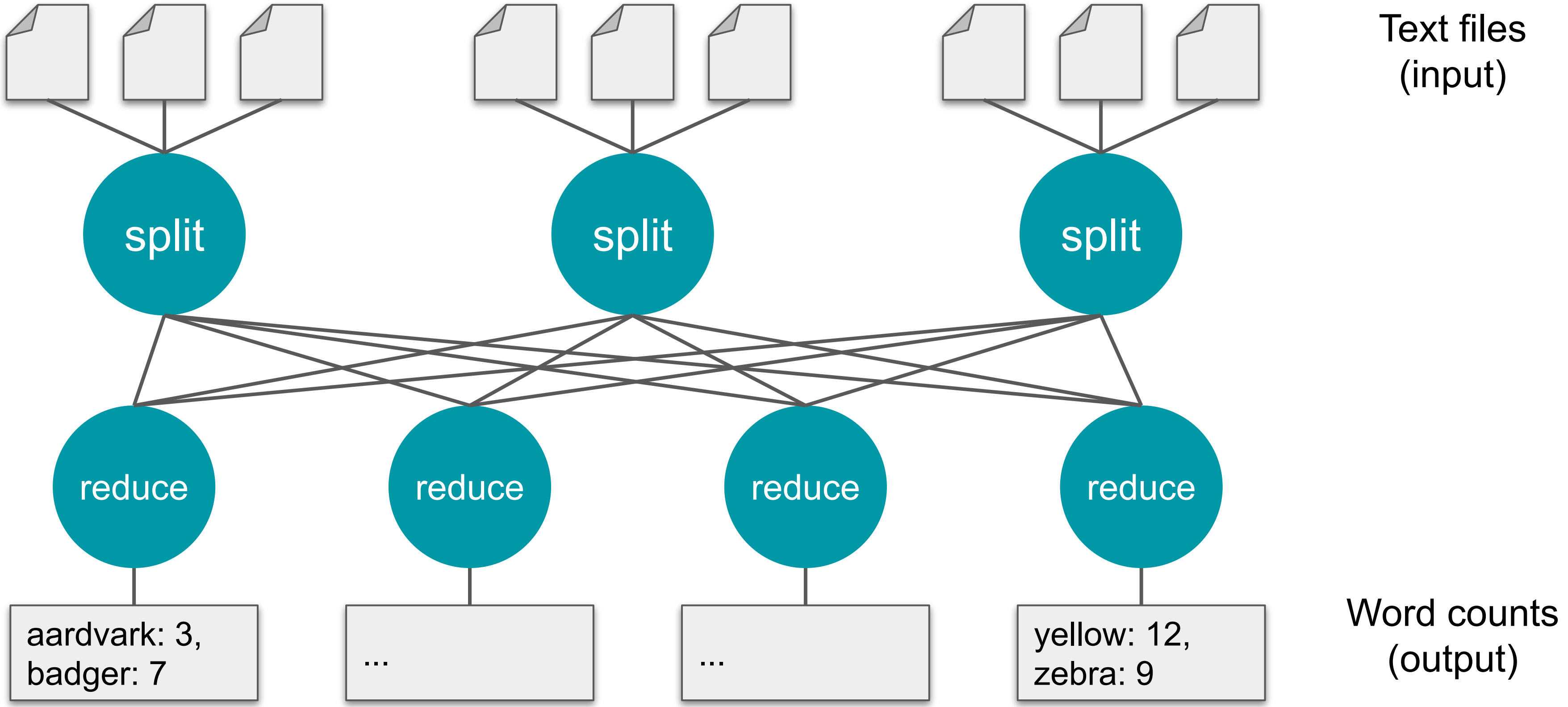
Building a document index



Building a document index



WordCount


Text files
(input)
split
split
split
reduce
reduce
reduce
reduce
aardvark: 3,
badger: 7
...
...
yellow: 12,
zebra: 9
Word counts
(output)

Data-parallel programming

Batch processing

Input is a large, immutable data sets

Processing can be processed in parallel pipelines

Significant chance of hardware failure during execution

Related programming models:

- Stream and incremental processing
- Deductive and graph programming

Data-parallel programming

Data-parallel framework (e.g. MapReduce, FlumeJava, Apache Spark)

```
(*) You write two small functions
fun wc_mapper line =
  List.map (fn w => (w, 1)) (split line);

fun wc_reducer (key, values) =
  List.foldl (fn (x, y) => x + y) 0 values;

(*) The framework provides mapReduce
fun mapReduce mapper reducer list = ...;

(*) Combine them to build your program
fun wordCount list = mapReduce wc_mapper wc_reducer list;
```

Data-parallel programming

Data-parallel framework (e.g. MapReduce, FlumeJava, Apache Spark)

```
(*) You write two small functions
fun wc_mapper line =
  List.map (fn w => (w, 1)) (split line);

fun wc_reducer (key, values) =
  List.foldl (fn (x, y) => x + y) 0 values;

(*) The framework provides mapReduce
fun mapReduce mapper reducer list = ...;

(*) Combine them to build your program
fun wordCount list = mapReduce wc_mapper wc_reducer list;
```

SQL

```
SELECT word, COUNT(*) AS c
FROM Documents AS d,
  LATERAL TABLE (split(d.text)) AS word   // requires a ‘split’ UDF
GROUP BY word;
```

Data-parallel programming

Data-parallel framework (e.g. MapReduce, FlumeJava, Apache Spark)

```
(*) You write two small functions
fun wc_mapper line =
  List.map (fn w => (w, 1)) (split line);

fun wc_reducer (key, values) =
  List.foldl (fn (x, y) => x + y) 0 values;

(*) The framework provides mapReduce
fun mapReduce mapper reducer list = ...;

(*) Combine them to build your program
fun wordCount list = mapReduce wc_mapper wc_reducer list;
```

SQL

```
SELECT word, COUNT(*) AS c
FROM Documents AS d,
  LATERAL TABLE (split(d.text)) AS word   // requires a ‘split’ UDF
GROUP BY word;
```

Morel

```
from line in lines,
    word in split line    (*) requires ‘split’ function - see later...
group word compute c = count
```

What are our options?

Extend SQL

We’ll need to:

- Allow functions defined in queries
- Add relations-as-values
- Add functions-as-values
- Modernize the type system (adding type variables, function types, algebraic types)
- Write an optimizing compiler

Extend a functional programming language

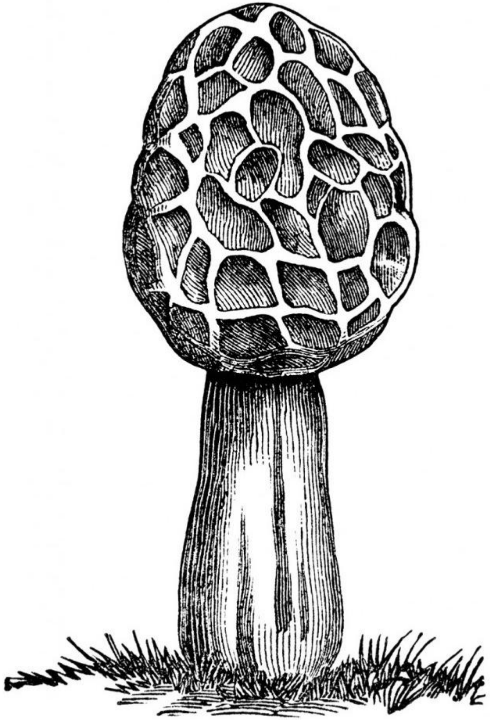
We’ll need to:

- Add syntax for relational operations
- Map onto external data
- Write a query optimizer

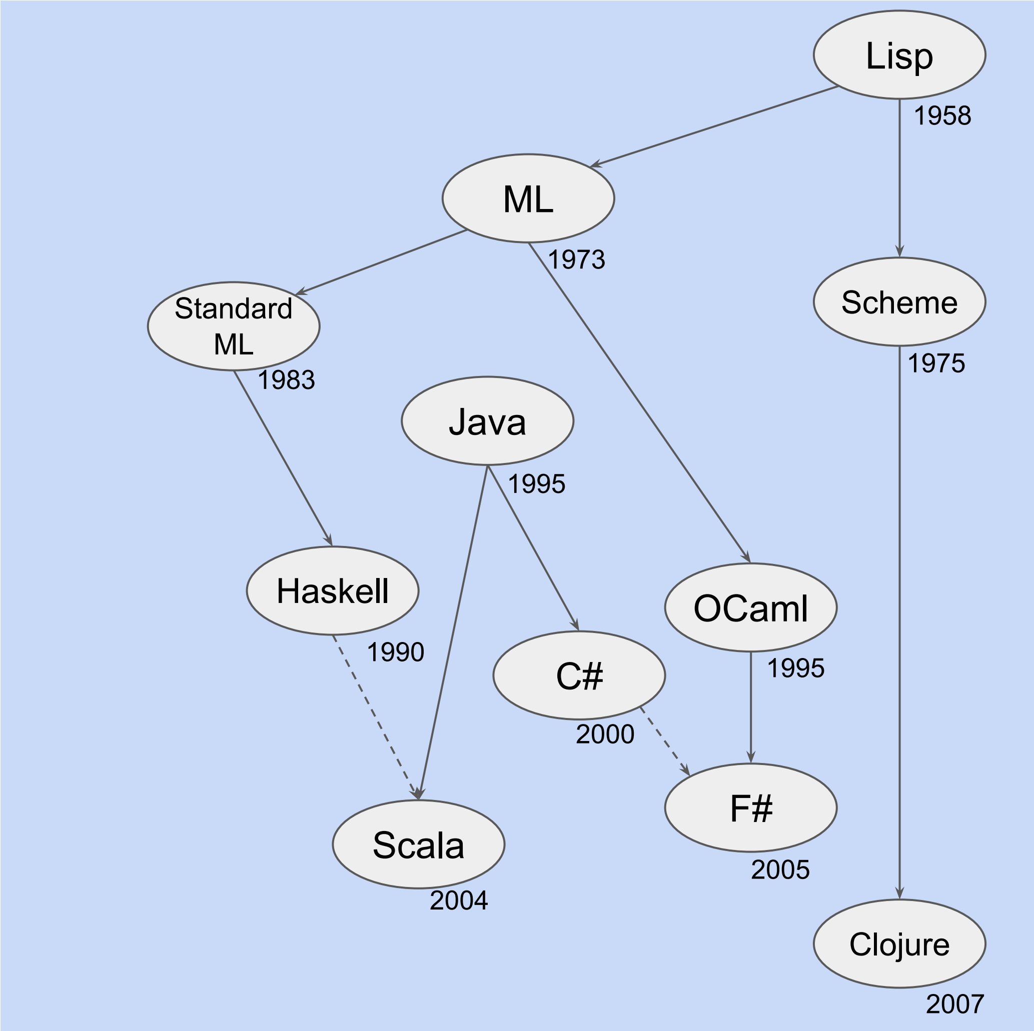
Nice stuff we get for free:

- Algebraic types
- Pattern matching
- Inline function and value declarations

Morel is a functional programming language. It is derived from Standard ML, and is extended with list comprehensions and other relational operators. Like Standard ML, Morel has parametric and algebraic data types with Hindley-Milner type inference. Morel is implemented in Java, and is optimized and executed using a combination of techniques from functional programming language compilers and database query optimizers.



Evolution of functional languages



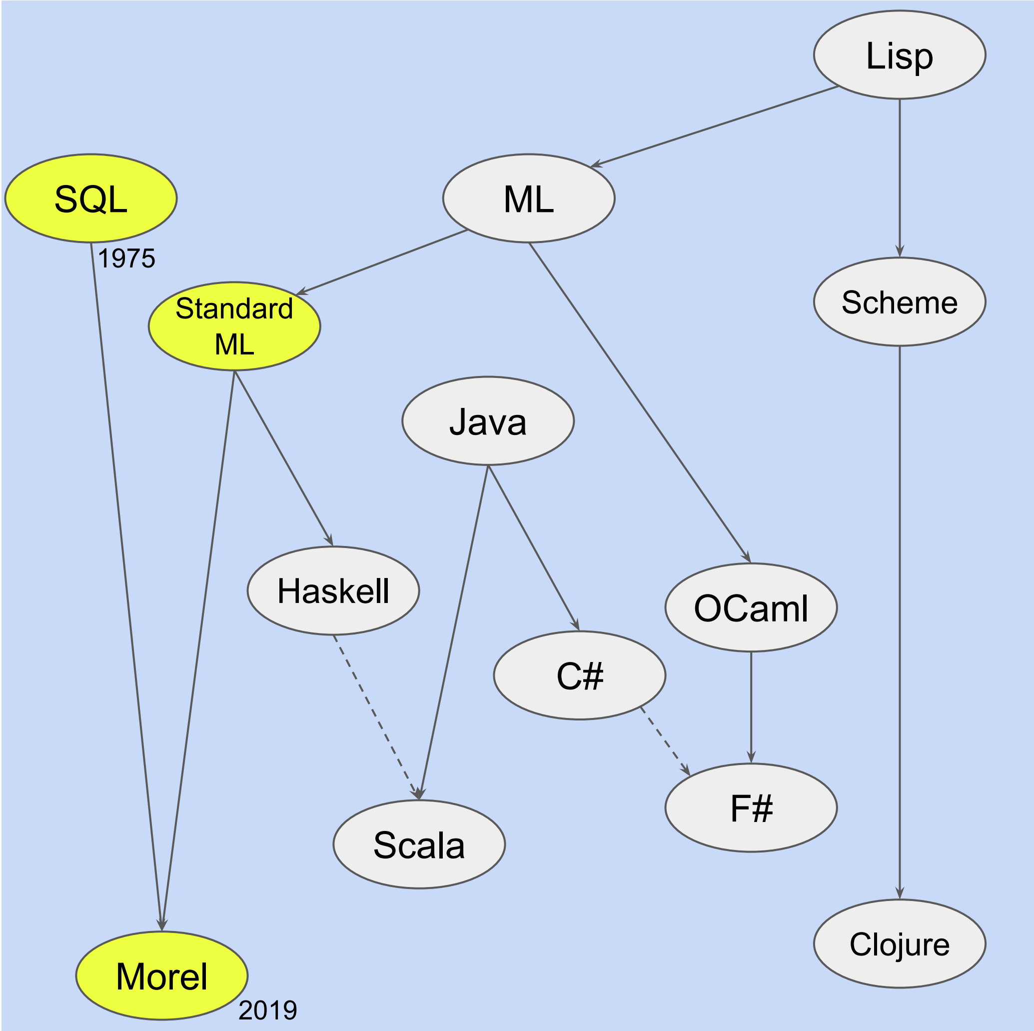
Extend Standard ML

Standard ML is strongly typed, and can very frequently deduce every type in a program.

Standard ML has record types. (These are important for queries.)

Haven't decided whether Morel is eager (like Standard ML) or lazy (like Haskell and SQL)

Haskell's type system is more powerful. So, Haskell programs often have explicit types.



Morel themes

Early stage language

The target audience is SQL users

Less emphasis on abstract algebra

Program compilation, query optimization

Functional programming in-the-small, and in-the-large

2. Functional

A quick introduction to Standard ML

All of the following examples work in both Standard ML and Morel.

Data parallel
(e.g. Flume, Spark)

Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

Standard ML: values and types

```
- "Hello, world!";
val it = "Hello, world!" : string

- 1 + 2;
val it = 3 : int

- ~1.5;
val it = ~1.5 : real

- [1, 1, 2, 3, 5];
val it = [1,1,2,3,5] : int list

- fn i => i mod 2 = 1;
val it = fn : int -> bool

- (1, "a");
val it = (1,"a") : int * string

- {name = "Fred", empno = 100};
val it = {empno=100,name="Fred"} : {empno:int, name:string}
```

Standard ML: variables and functions

```
- val x = 1;
val x = 1 : int

- val isOdd = fn i => i mod 2 = 1;
val isOdd = fn : int -> bool

- fun isOdd i = i mod 2 = 0;
val isOdd = fn : int -> bool

- isOdd x;
val it = true : bool

- let
=   val x = 6
=   fun isOdd i = i mod 2 = 1
= in
=   isOdd x
= end;
val it = false : bool
```

`val` assigns a value to a variable.

`fun` declares a function.

- `fun` is syntactic sugar for
  `val ... = fn ... => ...`

`let` allows you to make several declarations before evaluating an expression.

Algebraic data types, `case`, and recursion

`datatype` declares an algebraic data type. **`'a`** Is a type variable and therefore **`'a tree`** is a polymorphic type.

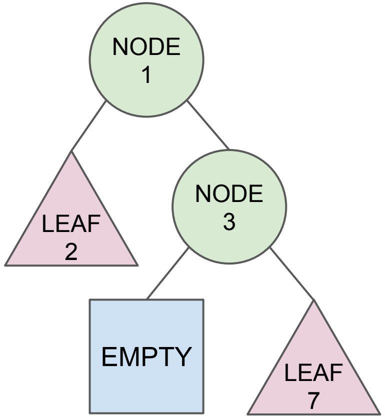
```
- datatype 'a tree =
=     EMPTY
=   | LEAF of 'a
=   | NODE of ('a * 'a tree * 'a tree);
```

Algebraic data types, `case`, and recursion

datatype declares an algebraic data type. **'a** Is a type variable and therefore **'a tree** is a polymorphic type.

```
- datatype 'a tree =
=       EMPTY
=     | LEAF of 'a
=     | NODE of ('a * 'a tree * 'a tree);

- val t =
=   NODE (1, LEAF 2, NODE (3, EMPTY, LEAF 7));
```

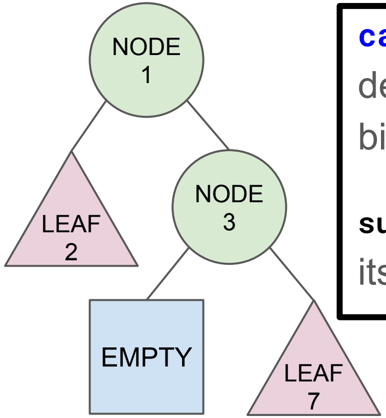
Define an instance of **tree** using its constructors **NODE**, **LEAF** and **EMPTY**.



Algebraic data types, `case`, and recursion

`datatype` declares an algebraic data type. `'a` Is a type variable and therefore `'a tree` is a polymorphic type.

Define an instance of `tree` using its constructors `NODE`, `LEAF` and `EMPTY`.



`case` matches patterns, deconstructing data types, and binding variables as it goes.

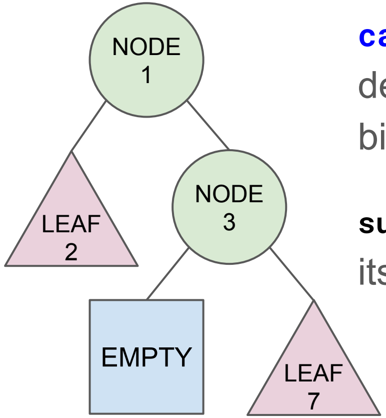
`sumTree` use `case` and calls itself recursively.

```
- datatype 'a tree =
=     EMPTY
=   | LEAF of 'a
=   | NODE of ('a * 'a tree * 'a tree);

- val t =
=   NODE (1, LEAF 2, NODE (3, EMPTY, LEAF 7));

- val rec sumTree = fn t =>
=   case t of EMPTY => 0
=     | LEAF i => i
=     | NODE (i, l, r) =>
=         i + sumTree l + sumTree r;
val sumTree = fn : int tree -> int

- sumTree t;
val it = 13 : int
```

Algebraic data types, case, and recursion

datatype declares an algebraic data type. `'a` Is a type variable and therefore `'a tree` is a polymorphic type.

Define an instance of `tree` using its constructors `NODE`, `LEAF` and `EMPTY`.



case matches patterns, deconstructing data types, and binding variables as it goes.

`sumTree` use **case** and calls itself recursively.

fun is a shorthand for **case** and **val rec**.

```
- datatype 'a tree =
=     EMPTY
=   | LEAF of 'a
=   | NODE of ('a * 'a tree * 'a tree);

- val t =
=   NODE (1, LEAF 2, NODE (3, EMPTY, LEAF 7));

- val rec sumTree = fn t =>
=   case t of EMPTY => 0
=     | LEAF i => i
=     | NODE (i, l, r) =>
=         i + sumTree l + sumTree r;
val sumTree = fn : int tree -> int

- sumTree t;
val it = 13 : int

- fun sumTree EMPTY = 0
=   | sumTree (LEAF i) = i
=   | sumTree (NODE (i, l, r)) =
=         i + sumTree l + sumTree r;
val sumTree = fn : int tree -> int
```

Relations and higher-order functions

Relations are lists of records.

A higher-order function is a function whose arguments or return value are functions.

Common higher-order functions that act on lists:

- `List.filter` – equivalent to the Filter relational operator (SQL `WHERE`)
- `List.map` – equivalent to Project relational operator (SQL `SELECT`)
- `flatMap` – as `List.map`, but may produce several output elements per input element

`#deptno` is a system-generated function that accesses the `deptno` field of a record.

```
- val emps = [
=   {id = 100, name = "Fred", deptno = 10},
=   {id = 101, name = "Velma", deptno = 20},
=   {id = 102, name = "Shaggy", deptno = 30},
=   {id = 103, name = "Scooby", deptno = 30}];

- List.filter (fn e => #deptno e = 30) emps;
val it = [{deptno=30,id=102,name="Shaggy"},
  {deptno=30,id=103,name="Scooby"}]
  : {deptno:int, id:int, name:string} list
```

Equivalent SQL

```
SELECT *
FROM emps AS e
WHERE e.deptno = 30
```

Implementing Join using higher-order functions

`List.map` function is equivalent to Project relational operator (SQL `SELECT`)

We also define a `flatMap` function.

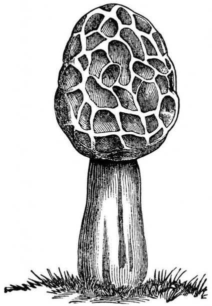
Equivalent SQL

```
SELECT d.deptno, e.name
FROM emps AS e,
  depts AS d
WHERE d.deptno = e.deptno
```

```
- val depts = [
=   {deptno = 10, name = "Sales"},
=   {deptno = 20, name = "Marketing"},
=   {deptno = 30, name = "R&D"}];

- fun flatMap f xs = List.concat (List.map f xs);
val flatMap =
  fn : ('a -> 'b list) -> 'a list -> 'b list

- List.map
=   (fn (d, e) => {deptno = #deptno d, name = #name e})
=   (List.filter
=     (fn (d, e) => #deptno d = #deptno e)
=     (flatMap
=       (fn e => (List.map (fn d => (d, e)) depts))
=       emps));
val it =
  [{deptno=10,name="Fred"}, {deptno=20,name="Velma"},
   {deptno=30,name="Shaggy"}, {deptno=30,name="Scooby"}]
 : {deptno:int, name:string} list
```

3. Functional + Relational

Morel



Data parallel
(e.g. Flume, Spark)

Relational
(e.g. SQL)

Deductive
(e.g. Datalog)

Functional
(e.g. Haskell, Standard ML, Morel)

Implementing Join in Morel using `from`

Morel extensions to Standard ML:

- `from` operator creates a list comprehension
- `x.field` is shorthand for `#field x`
- `{#field}` is shorthand for `{field = #field x}`

```
List.map
  (fn (d, e) => {deptno = #deptno d, name = #name e})
  (List.filter
    (fn (d, e) => #deptno d = #deptno e)
    (flatMap
      (fn e => (List.map (fn d => (d, e)) depts))
      emps));
```

Equivalent SQL

```
SELECT d.deptno, e.name
FROM emps AS e,
  depts AS d
WHERE d.deptno = e.deptno
```

Implementing Join in Morel using `from`

Morel extensions to Standard ML:

- `from` operator creates a list comprehension
- `x.field` is shorthand for `#field x`
- `{#field}` is shorthand for `{field = #field x}`

```
List.map
  (fn (d, e) => {deptno = #deptno d, name = #name e})
  (List.filter
    (fn (d, e) => #deptno d = #deptno e)
    (flatMap
      (fn e => (List.map (fn d => (d, e)) depts))
      emps));
```

```
from e in emps,
    d in depts
  where #deptno e = #deptno d
  yield {deptno = #deptno d, name = #name e};
```

Equivalent SQL

```
SELECT d.deptno, e.name
FROM emps AS e,
  depts AS d
WHERE d.deptno = e.deptno
```

Implementing Join in Morel using `from`

Morel extensions to Standard ML:

- `from` operator creates a list comprehension
- `x.field` is shorthand for `#field x`
- `{#field}` is shorthand for `{field = #field x}`

```
List.map
  (fn (d, e) => {deptno = #deptno d, name = #name e})
  (List.filter
    (fn (d, e) => #deptno d = #deptno e)
    (flatMap
      (fn e => (List.map (fn d => (d, e)) depts))
      emps));
```

```
from e in emps,
    d in depts
  where #deptno e = #deptno d
  yield {deptno = #deptno d, name = #name e};
```

```
from e in emps,
    d in depts
  where e.deptno = d.deptno
  yield {d.deptno, e.name};
```

Equivalent SQL

```
SELECT d.deptno, e.name
FROM emps AS e,
  depts AS d
WHERE d.deptno = e.deptno
```

WordCount

```
let
  fun split0 [] word words = word :: words
    | split0 (#" " :: s) word words = split0 s "" (word :: words)
    | split0 (c :: s) word words = split0 s (word ^ (String.str c)) words
  fun split = List.rev (split0 (String.explode s) "" [])
in
  from line in lines,
    word in split line
  group word compute c = count
end;
```

WordCount

```
- let
=   fun split0 [] word words = word :: words
=     | split0 (#" " :: s) word words = split0 s "" (word :: words)
=     | split0 (c :: s) word words = split0 s (word ^ (String.str c)) words
=   fun split = List.rev (split0 (String.explode s) "" [])
= in
=   from line in lines,
=     word in split line
=   group word compute c = count
= end;
val wordCount = fn : string list -> {c:int, word:string} list

- wordCount ["a skunk sat on a stump",
=     "and thunk the stump stunk",
=     "but the stump thunk the skunk stunk"];
val it =
  [{c=2,word="a"},{c=3,word="the"},{c=1,word="but"},
   {c=1,word="sat"},{c=1,word="and"},{c=2,word="stunk"},
   {c=3,word="stump"},{c=1,word="on"},{c=2,word="thunk"},
   {c=2,word="skunk"}] : {c:int, word:string} list
```

WordCount

Functional programming “in the small”

```
- let
=   fun split0 [] word words = word :: words
=     | split0 (#" " :: s) word words = split0 s "" (word :: words)
=     | split0 (c :: s) word words = split0 s (word ^ (String.str c)) words
=   fun split = List.rev (split0 (String.explode s) "" [])
= in
=   from line in lines,
=     word in split line
=   group word compute c = count
= end;
val wordCount = fn : string list -> {c:int, word:string} list

- wordCount ["a skunk sat on a stump",
=     "and thunk the stump stunk",
=     "but the stump thunk the skunk stunk"];
val it =
  [{c=2,word="a"},{c=3,word="the"},{c=1,word="but"},
   {c=1,word="sat"},{c=1,word="and"},{c=2,word="stunk"},
   {c=3,word="stump"},{c=1,word="on"},{c=2,word="thunk"},
   {c=2,word="skunk"}] : {c:int, word:string} list
```

WordCount

```
- let
=   fun split0 [] word words = word :: words
=     | split0 (#" " :: s) word words = split0 s "" (word :: words)
=     | split0 (c :: s) word words = split0 s (word ^ (String.str c)) words
=   fun split = List.rev (split0 (String.explode s) "" [])
= in
=   from line in lines,
=     word in split line
=   group word compute c = count
= end;
val wordCount = fn : string list -> {c:int, word:string} list

- wordCount ["a skunk sat on a stump",
=     "and thunk the stump stunk",
=     "but the stump thunk the skunk stunk"];
val it =
  [{c=2,word="a"},{c=3,word="the"},{c=1,word="but"},
   {c=1,word="sat"},{c=1,word="and"},{c=2,word="stunk"},
   {c=3,word="stump"},{c=1,word="on"},{c=2,word="thunk"},
   {c=2,word="skunk"}] : {c:int, word:string} list
```

Functional programming “in the large”

Functions as views, functions as values

```
- fun emps2 () =
=     from e in emps
=       yield {e.id,
=         e.name,
=         e.deptno,
=         comp = fn revenue => case e.deptno of
=             30 => e.id + revenue / 2
=           | _ => e.id};
val emps2 = fn : unit -> {comp:int -> int,
    deptno:int, id:int, name:string} list
```

Functions as views, functions as values

emps2 is a function that returns a collection

```
- fun emps2 () =
=     from e in emps
=       yield {e.id,
=         e.name,
=         e.deptno,
=         comp = fn revenue => case e.deptno of
=             30 => e.id + revenue / 2
=           | _ => e.id};
val emps2 = fn : unit -> {comp:int -> int,
    deptno:int, id:int, name:string} list
```

Functions as views, functions as values

```
- fun emps2 () =
=     from e in emps
=       yield {e.id,
=         e.name,
=         e.deptno,
=         comp = fn revenue => case e.deptno of
=             30 => e.id + revenue / 2
=           | _ => e.id};
val emps2 = fn : unit -> {comp:int -> int,
    deptno:int, id:int, name:string} list
```

emps2 is a function that returns a collection

The comp field is a function value (in fact, it's a closure)

Functions as views, functions as values

emps2 is a function that returns a collection

```
- fun emps2 () =
=     from e in emps
=       yield {e.id,
=         e.name,
=         e.deptno,
=         comp = fn revenue => case e.deptno of
=             30 => e.id + revenue / 2
=           | _ => e.id};
val emps2 = fn : unit -> {comp:int -> int,
    deptno:int, id:int, name:string} list

- from e in emps2 ()
=   yield {e.name, e.id, c = e.comp 1000};
```

The comp field is a function value (in fact, it's a closure)

Functions as views, functions as values

```
- fun emps2 () =
=     from e in emps
=       yield {e.id,
=         e.name,
=         e.deptno,
=         comp = fn revenue => case e.deptno of
=             30 => e.id + revenue / 2
=           | _ => e.id};
val emps2 = fn : unit -> {comp:int -> int,
    deptno:int, id:int, name:string} list

- from e in emps2 ()
=   yield {e.name, e.id, c = e.comp 1000};
val it =
  [{c=100,id=100,name="Fred"},
   {c=101,id=101,name="Velma"},
   {c=602,id=102,name="Shaggy"},
   {c=603,id=103,name="Scooby"}]
  : {c:int, id:int, name:string} list
```

Chaining relational operators

```
- from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
=   where nameLength > 4
=   group deptno compute c = count, s = sum of nameLength
=   where s > 10
=   yield c + s;

val it = [14] : int list
```

Chaining relational operators - step 1

```
- from e in emps;






val it =
  [{deptno=10,id=100,name="Fred"},
   {deptno=20,id=101,name="Velma"},
   {deptno=30,id=102,name="Shaggy"},
   {deptno=30,id=103,name="Scooby"}]
  : {deptno:int, id:int, name:string} list
```

Chaining relational operators - step 2

```
- from e in emps
=   order e.deptno, e.id desc;






val it =
  [{deptno=10,id=100,name="Fred"},
   {deptno=20,id=101,name="Velma"},
   {deptno=30,id=103,name="Scooby"},
   {deptno=30,id=102,name="Shaggy"}]
  : {deptno:int, id:int, name:string} list
```

Chaining relational operators - step 3

```
- from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno};




val it =
  [{deptno=10,id=100,name="Fred",nameLength=4},
   {deptno=20,id=101,name="Velma",nameLength=5},
   {deptno=30,id=103,name="Scooby",nameLength=6},
   {deptno=30,id=102,name="Shaggy",nameLength=6}]
  : {deptno:int, id:int, name:string, nameLength:int} list
```

Chaining relational operators - step 4

```
- from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
=   where nameLength > 4;




val it =
  [{deptno=20,id=101,name="Velma",nameLength=5},
   {deptno=30,id=103,name="Scooby",nameLength=6},
   {deptno=30,id=102,name="Shaggy",nameLength=6}]
  : {deptno:int, id:int, name:string, nameLength:int} list
```

Chaining relational operators - step 5

```
- from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
=   where nameLength > 4
=   group deptno compute c = count, s = sum of nameLength;



val it =
  [{c=1,deptno=20,s=5},
   {c=2,deptno=30,s=12}]
  : {c:int, deptno:int, s:int} list
```

Chaining relational operators - step 6

```
- from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
=   where nameLength > 4
=   group deptno compute c = count, s = sum of nameLength
=   where s > 10;


val it =
  [{c=2,deptno=30,s=12}]
  : {c:int, deptno:int, s:int} list
```

Chaining relational operators

```
= from e in emps
=   order e.deptno, e.id desc
=   yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
=   where nameLength > 4
=   group deptno compute c = count, s = sum of nameLength
=   where s > 10
=   yield c + s;

val it = [14] : int list
```

Chaining relational operators

Morel

```
from e in emps
  order e.deptno, e.id desc
  yield {e.name, nameLength = String.size e.name, e.id, e.deptno}
  where nameLength > 4
  group deptno compute c = count, s = sum of nameLength
  where s > 10
  yield c + s;
```

SQL (almost equivalent)

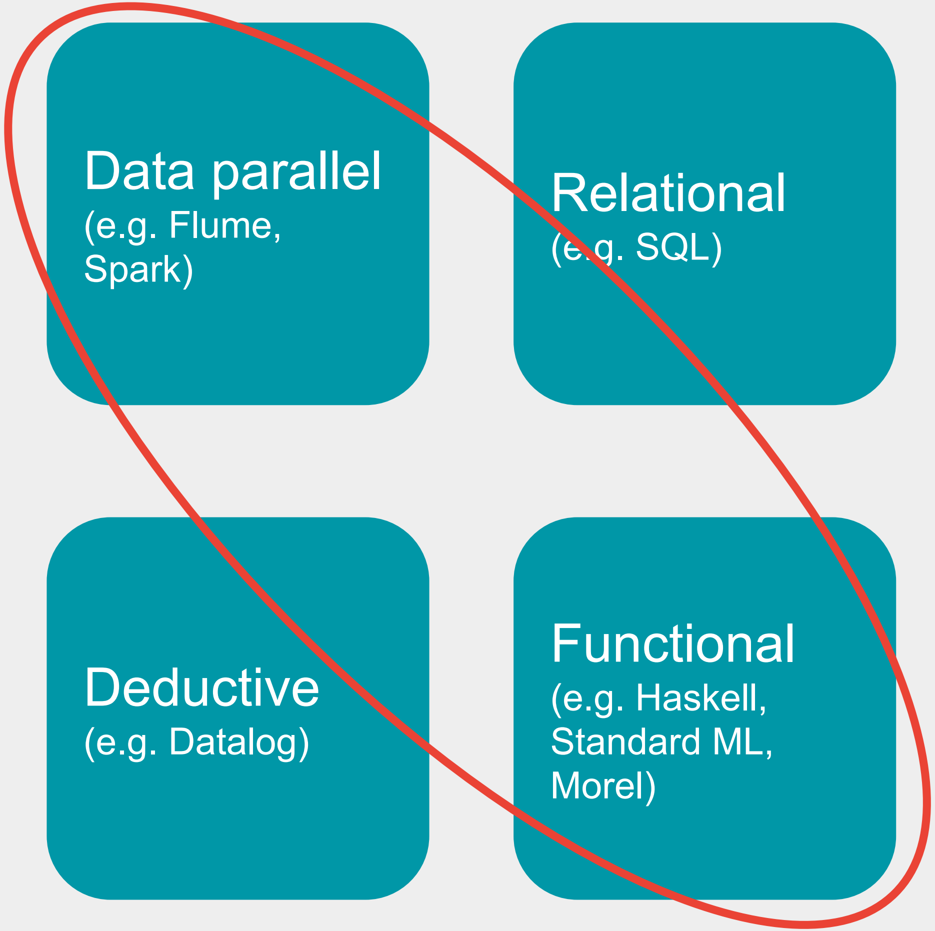
```
SELECT c + s
FROM (SELECT deptno, COUNT(*) AS c, SUM(nameLength) AS s
  FROM (SELECT e.name, CHAR_LENGTH(e.ename) AS nameLength, e.id, e.deptno
    FROM (SELECT *
      FROM emps AS e
      ORDER BY e.deptno, e.id DESC))
  WHERE nameLength > 4
  GROUP BY deptno
  HAVING s > 10)
```

Java (very approximately)

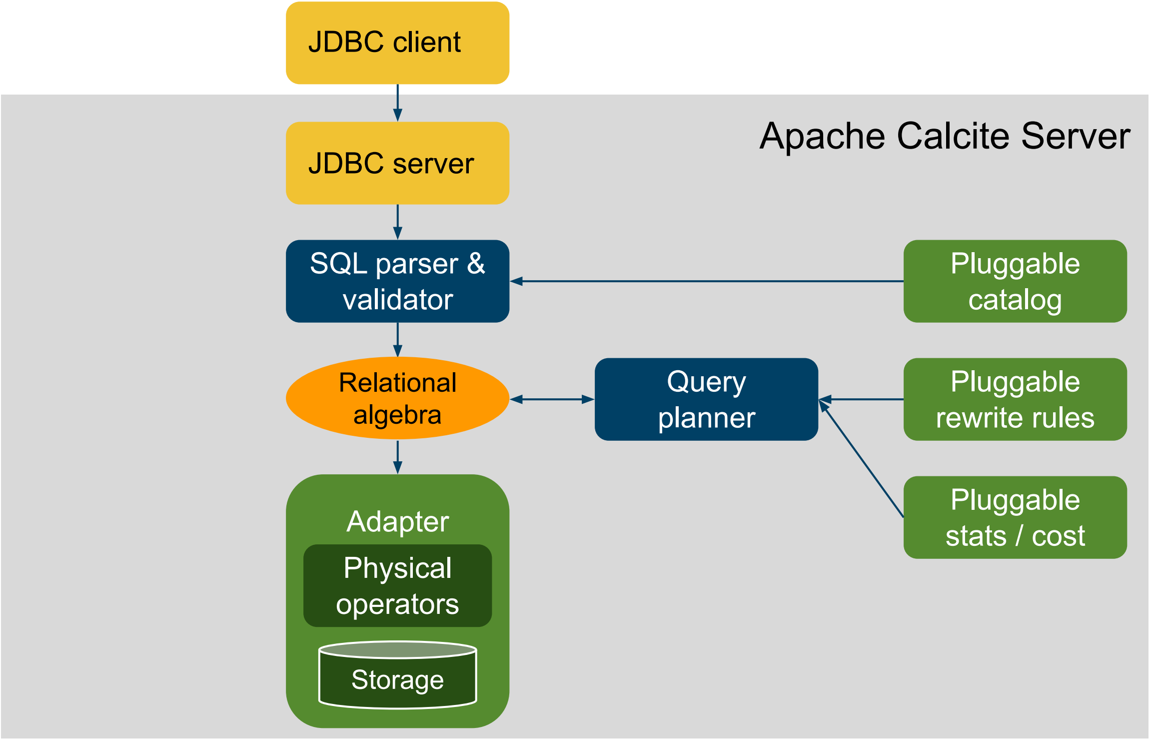
```
for (Emp e : emps) {
  String name = e.name;
  int nameLength = name.length();
  int id = e.id;
  int deptno = e.deptno;
  if (nameLength > 4) {
    ...
```

4. Functional + Data parallel

Integrating Morel with Apache Calcite



Apache Calcite



Toolkit for writing a DBMS

Many parts are optional or pluggable

Relational algebra is the core

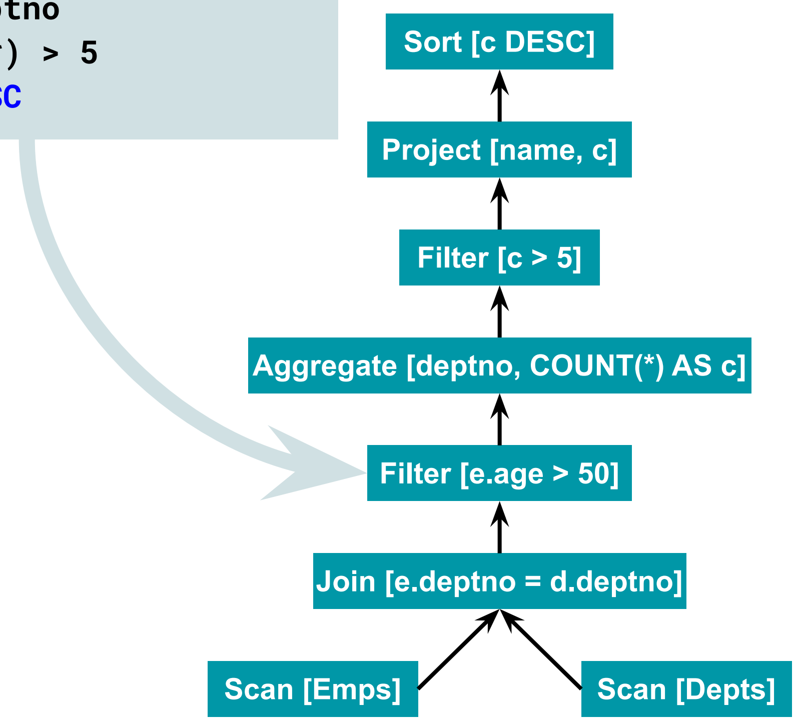
Relational algebra

Based on set theory, plus operators: Project, Filter, Aggregate, Union, Join, Sort

Calcite provides:

- SQL to relational algebra
- Query planner
- Physical operators to execute plan
- An adapter system to make external data sources look like tables

```
SELECT d.name, COUNT(*) AS c
FROM Emps AS e
JOIN Depts AS d USING (deptno)
WHERE e.age > 50
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```

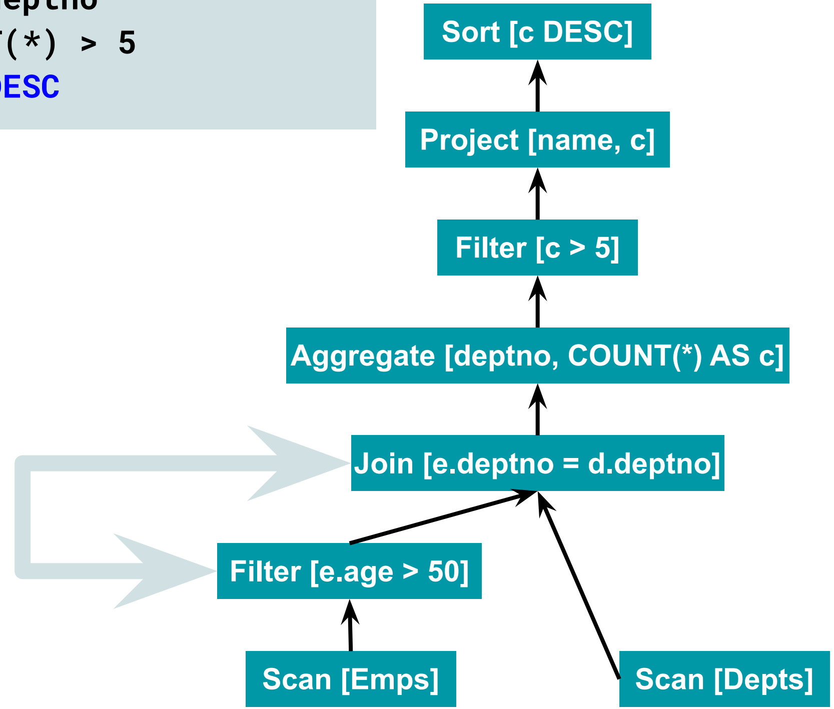


Algebraic rewrite

Calcite optimizes queries by applying rewrite rules that preserve semantics.

Planner uses dynamic programming, seeking the lowest total cost.

```sql
SELECT d.name, COUNT(*) AS c
FROM (SELECT * FROM Emps
      WHERE e.age > 50) AS e
JOIN Depts AS d USING (deptno)
GROUP BY d.deptno
HAVING COUNT(*) > 5
ORDER BY c DESC
```



Integration with Apache Calcite - schemas

Expose Calcite schemas as named records, each field of which is a table.

A default Morel connection has variables `foodmart` and `scott` (connections to hsqldb via Calcite's JDBC adapter).

Connections to other Calcite adapters (Apache Cassandra, Druid, Kafka, Pig, Redis) are also possible.

```
- foodmart;
val it = {account=<relation>, currency=<relation>,
  customer=<relation>, ...} : ...;

- scott;
val it = {bonus=<relation>, dept=<relation>,
  emp=<relation>, salgrade=<relation>} : ...;

- scott.dept;
val it =
  [{deptno=10,dname="ACCOUNTING",loc="NEW YORK"},
   {deptno=20,dname="RESEARCH",loc="DALLAS"},
   {deptno=30,dname="SALES",loc="CHICAGO"},
   {deptno=40,dname="OPERATIONS",loc="BOSTON"}]
  : {deptno:int, dname:string, loc:string} list

- from d in scott.dept
=   where notExists (from e in scott.emp
=     where e.deptno = d.deptno
=     andalso e.job = "CLERK");
val it =
  [{deptno=40,dname="OPERATIONS",loc="BOSTON"}]
  : {deptno:int, dname:string, loc:string} list
```

Integration with Apache Calcite - relational algebra

```
- Sys.set ("hybrid", true);
val it = () : unit

- from d in scott.dept
=   where notExists (from e in scott.emp
=     where e.deptno = d.deptno
=     andalso e.job = "CLERK");
val it = [{deptno=40,dname="OPERATIONS",loc="BOSTON"}]
  : {deptno:int, dname:string, loc:string} list
```

In “hybrid” mode, Morel’s compiler identifies sections of Morel programs that are relational operations and converts them to Calcite relational algebra.

Calcite can them optimize these and execute them on their native systems.

Integration with Apache Calcite - relational algebra

```
- Sys.set ("hybrid", true);
val it = () : unit

- from d in scott.dept
=   where notExists (from e in scott.emp
=     where e.deptno = d.deptno
=     andalso e.job = "CLERK");
val it = [{deptno=40,dname="OPERATIONS",loc="BOSTON"}]
  : {deptno:int, dname:string, loc:string} list

- Sys.plan();
val it = "calcite(plan
LogicalProject(deptno=[$0], dname=[$1], loc=[$2])
  LogicalFilter(condition=[IS NULL($4)])
    LogicalJoin(condition=[=($0, $3)], joinType=[left])
      LogicalProject(deptno=[$0], dname=[$1], loc=[$2])
        JdbcTableScan(table=[[scott, DEPT]])
      LogicalProject(deptno=[$0], $f1=[true])
        LogicalAggregate(group=[{0}])
          LogicalProject(deptno=[$1])
            LogicalFilter(condition=[AND(=($5, 'CLERK'), IS NOT NULL($1))])
              LogicalProject(comm=[$6], deptno=[$7], empno=[$0],ename=[$1], hiredate=[$4], job=[$2]
                JdbcTableScan(table=[[scott, EMP]]))" : string
```

In “hybrid” mode, Morel’s compiler identifies sections of Morel programs that are relational operations and converts them to Calcite relational algebra.

Calcite can them optimize these and execute them on their native systems.

Optimization

Relational query optimization

- Applies to relational operators (~10 core operators + UDFs) not general-purpose code
- Transformation rules that match patterns (e.g. Filter on Project)
- Decisions based on cost & statistics
- Hybrid plans - choose target engine, sort order
- Materialized views
- Macro-level optimizations

Functional program optimization

- Applies to typed lambda calculus
- Inline constant lambda expressions
- Eliminate dead code
- Defend against mutually recursive groups
- Careful to not inline expressions that are used more than once (increasing code size) or which bind closures more often (increasing running time)
- Micro-level optimizations

[GM] "The Volcano Optimizer Generator: Extensibility and Efficient Search" (Graefe, McKenna 1993)
[PJM] "Secrets of the Glasgow Haskell Compiler inliner" (Peyton Jones, Marlow 2002)

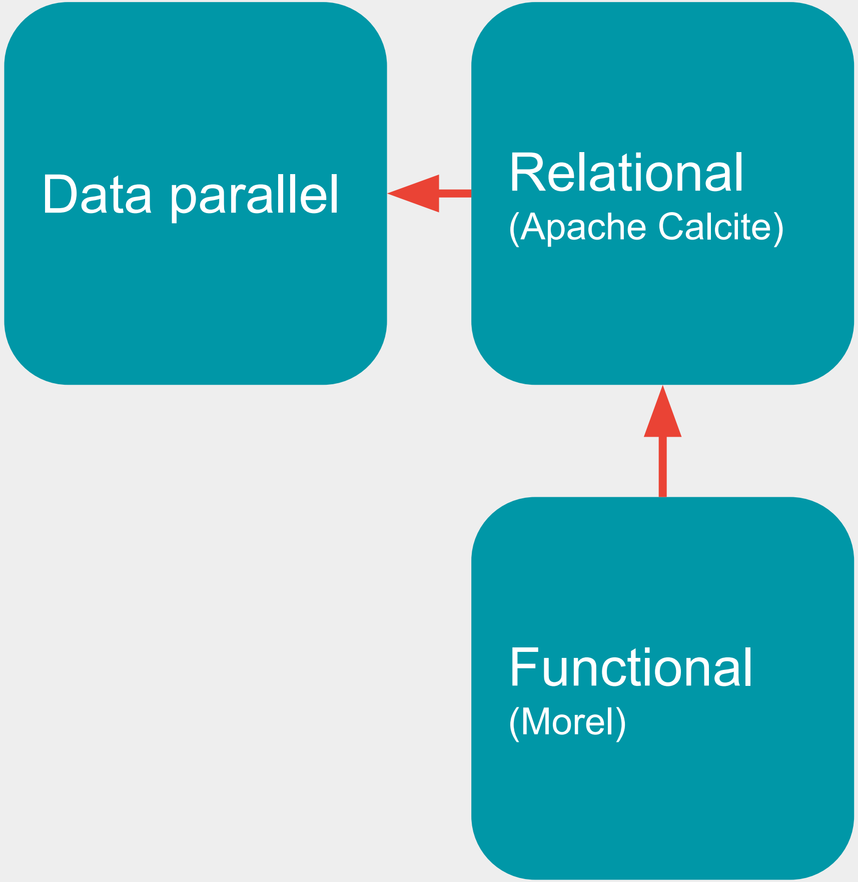
WordCount

```
from line in lines,
   word in split line
 group word compute c = count;
```

Concisely specified in a functional programming language (Morel)

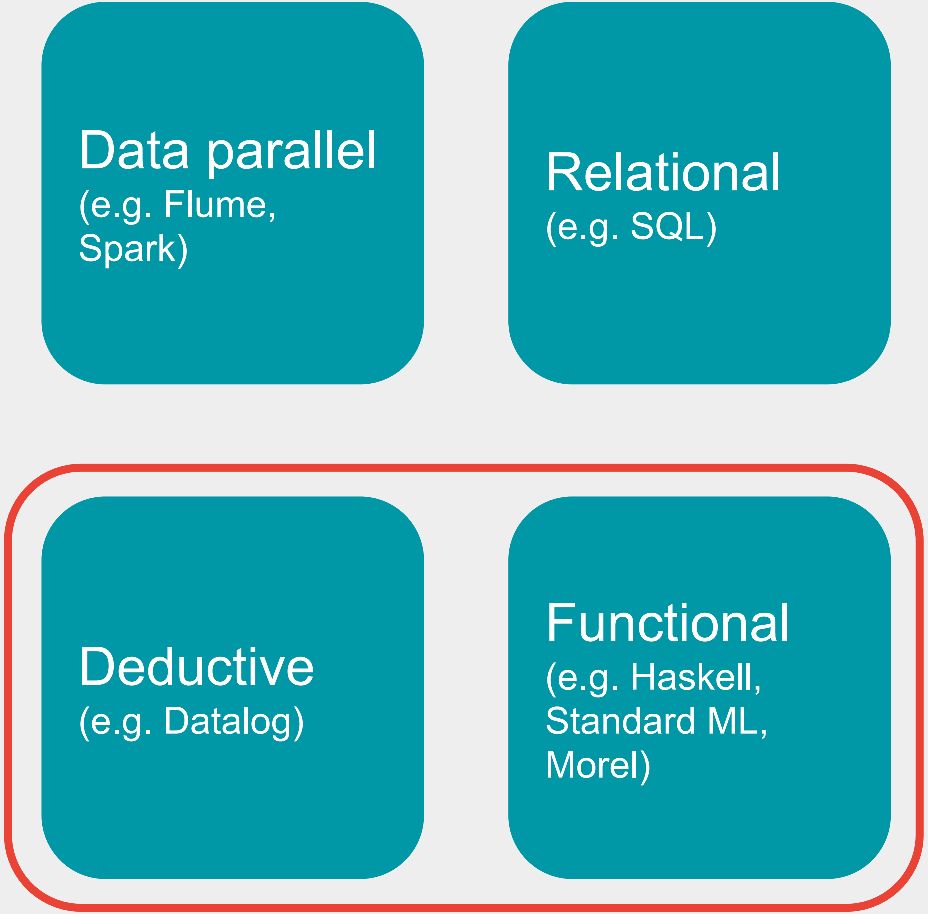
Translated into relational algebra (Apache Calcite)

Executed in the data-parallel framework of your choice



5. Functional + Deductive

An elegant way to recursively define relations in a functional programming language*



*Designed but not yet implemented

Parents (a base relation)

```
# SQL
SELECT *
FROM parents
WHERE parent = ‘elrond’;
parent child
====== =======
elrond arwen
elrond elladan
elrond elrohir
```

```
# Morel
from (parent, child) in parents
  where parent = “elrond”;
[(“elrond”, “arwen”),
 (“elrond”, “elladan”),
 (“elrond”, “elrohir”)];
```

```
# Datalog
is_parent(earendil, elrond).
is_parent(elrond, arwen).
is_parent(elrond, elladan).
is_parent(elrond, elrohir).

answer(X) :- is_parent(elrond, X).
X = arwen
X = elladan
X = elrohir
```

Ancestors (a recursively-defined relation)

```
# SQL
CREATE VIEW ancestors AS
  WITH RECURSIVE a AS (
    SELECT parent AS ancestor,
      child AS descendant
    FROM parents
  UNION ALL
    SELECT a.ancestor, p.child
    FROM parents AS p
    JOIN a ON a.descendant = p.parent)
  SELECT * FROM a;

SELECT * FROM ancestors
WHERE descendant = ‘arwen’;
ancestor descendant
======== =======
earendil arwen
elrond   arwen
```

```
# Datalog
is_ancestor(X, Y) :- is_parent(X, Y).
is_ancestor(X, Y) :- is_parent(X, Z),
                     is_ancestor(Z, Y).

answer(X) :- is_ancestor(X, arwen).
```

```
# Morel
fun ancestors () =
    (from (x, y) in parents)
    union
    (from (x, y) in parents,
        (y2, z) in ancestors ()
      where y = y2
      yield (x, z));

from (ancestor, descendant) in ancestors ()
  where descendant = “arwen”;
Uncaught exception: StackOverflowError
```

Ancestors (a recursively-defined relation)

```
# SQL
CREATE VIEW ancestors AS
  WITH RECURSIVE a AS (
    SELECT parent AS ancestor,
      child AS descendant
    FROM parents
  UNION ALL
    SELECT a.ancestor, p.child
    FROM parents AS p
    JOIN a ON a.descendan
  SELECT * FROM a;

SELECT * FROM ancestors
WHERE descendant = ‘arwen
ancestor descendant
======== =======
earendil arwen
elrond   arwen
```

“StackOverflowError” is not easy to fix. It’s hard to a write a recursive function that iterates until a set reaches a fixed point.

```
# Datalog
is_ancestor(X, Y) :- is_parent(X, Y).
is_ancestor(X, Y) :- is_parent(X, Z),
                     is_ancestor(Z, Y).

answer(X) :- is_ancestor(X, arwen).
```

```
# Morel
fun ancestors () =
    (from (x, y) in parents)
    union
    (from (x, y) in parents,
        (y2, z) in ancestors ()
      where y = y2
      yield (x, z));

from (ancestor, descendant) in ancestors ()
  where descendant = “arwen”;
Uncaught exception: StackOverflowError
```

Two ways to define a relation

```
# Morel “forwards” relation

# Relation defined using algebra
- fun clerks () =
=   from e in emps
=     where e.job = “CLERK”;

# Query uses regular iteration
- from e in clerks,
=     d in depts
=   where d.deptno = e.deptno
=   andalso d.loc = “DALLAS”
=   yield e.name;
val it =
  [“SMITH”, “ADAMS”] : string list;
```

```
# Morel “backwards” relation

# Relation defined using a predicate
- fun isClerk e =
=   e.job = “CLERK”;

# Query uses a mixture of constrained
# and regular iteration
- from e suchThat isClerk e,
=     d in depts
=   where d.deptno = e.deptno
=   andalso d.loc = “DALLAS”
=   yield e.name;
val it =
  [“SMITH”, “ADAMS”] : string list;
```

Two ways to define a relation

```
# Morel “forwards” relation

# Relation defined using algebra
- fun clerks () =
=   from e in emps
=     where e.job = “CLERK”;

# Query uses regular iteration
- from e in clerks,
=     d in depts
=   where d.deptno = e.deptno
=   andalso d.loc = “DALLAS”
=   yield e.name;
val it =
  [“SMITH”, “ADAMS”] : string list;
```

```
# Morel “backwards” relation

# Relation defined using a predicate
- fun isClerk e =
=   e.job = “CLERK”;

# Query uses a mixture of constrained
# and regular iteration
- from e suchThat isClerk e,
=     d in depts
=   where d.deptno    e.deptno
=   andalso d.loc =   ALLAS”
=   yield e.name;
val it =
  [“SMITH”, “ADAMS”
```

“suchThat” keyword converts a bool function into a relation

Recursively-defined predicate relation

```
# Morel
- fun isAncestor (x, z) =
=   (x, z) elem parents
=   orelse exists (
=     from y suchThat isAncestor (x, y)
=              andalso (y, z) elem parents);

- from a suchThat isAncestor (a, “arwen”);
val it = [“earendil”, “elrond”] : string list

- from d suchThat isAncestor (“earendil”, d);
val it = [“elrond”, “arwen”, “elladan”, “elrohir”] : string list
```

Morel

Functional query language

Rich type system, concise as SQL, Turing complete

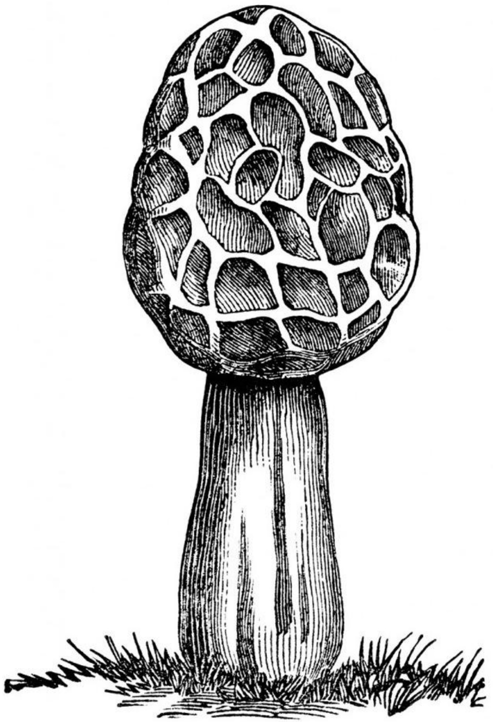
Combines (relational) query optimization with (lambda) FP optimization techniques

Execute in Java interpreter and/or data-parallel engines

```
from e in emps,
    d in depts
  where e.deptno = d.deptno
  yield {d.deptno, e.name};
```

```
from line in lines,
   word in split line
 group word compute c = count;
```

```
from a suchThat isAncestor (a, "arwen");
```



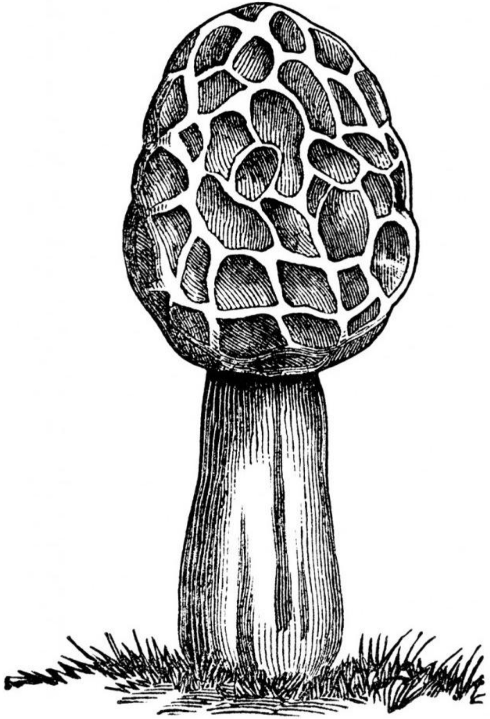
Thank you! Questions?

Julian Hyde, Looker/Google • @julianhyde

https://github.com/hydromatic/morel • @morel_lang

https://calcite.apache.org • @ApacheCalcite

ADCF FORUM



Extra slides

Other operators

from in

where

yield

order … desc

group … compute

count max min sum

intersect union except

exists notExists elem notElem only

Compared to other languages

Haskell – Haskell comprehensions are more general (monads vs lists). Morel is focused on relational algebra and probably benefits from a simpler type system.

Builder APIs (e.g. LINQ, FlumeJava, Cascading, Apache Spark) – Builder APIs are two languages. E.g. Apache Spark programs are Scala that builds an algebraic expression. Scala code (especially lambdas) is sprinkled throughout the algebra. Scala compilation is not integrated with algebra optimization.

SQL – SQL's type system does not have parameterized types, so higher order functions are awkward. Tables and columns have separate namespaces, which complicates handling of nested collections. Functions and temporary variables cannot be defined inside queries, so queries are not Turing-complete (unless you use recursive query gymnastics).

Standard ML: types

	Type	Example

Primitive types	`bool`
	`char`
	`int`
	`real`
	`string`
	`unit`
		`true: bool`
		`#"a": char`
		`~1: int`
		`3.14: real`
		`"foo": string`
		`(): unit`

Function types	`string -> int` `int * int -> int`	`String.size` `fn (x, y) => x + y * y`
Tuple types	`int * string`	`(10, "Fred")`
Record types	`{empno:int, name:string}`	`{empno=10, name="Fred"}`
Collection types	`int list` `(bool * (int -> int)) list`	`[1, 2, 3]` `[(true, fn i => i + 1)]`
Type variables	`'a`	`List.length: 'a list -> int`

Functional programming ↔ relational programming

Functional programming in-the-small

```
- fun squareList [] = []
=   | squareList (x :: xs) = x * x :: squareList xs;
val squareList = fn : int list -> int list

- squareList [1, 2, 3];
val it = [1,4,9] : int list
```

Functional programming in-the-large

```
- fun squareList xs = List.map (fn x => x * x) xs;
val squareList = fn : int list -> int list

- squareList [1, 2, 3];
val it = [1,4,9] : int list
```

Relational programming

```
- fun squareList xs =
=   from x in xs
=     yield x * x;

- squareList [1, 2, 3];
val it = [1,4,9] : int list
```

wordCount again

wordCount in-the-small

```
- fun wordCount list = ...;
val wordCount = fn : string list -> {count:int, word:string} list
```

wordCount in-the-large using `mapReduce`

```
- fun mapReduce mapper reducer list = ...;
val mapReduce = fn : ('a -> ('b * 'c) list) ->
    ('b * 'c list -> 'd) -> 'a list -> ('b * 'd) list

- fun wc_mapper line =
=   List.map (fn w => (w, 1)) (split line);
val wc_mapper = fn : string -> (string * int) list

- fun wc_reducer (key, values) =
=   List.foldl (fn (x, y) => x + y) 0 values;
val wc_reducer = fn : 'a * int list -> int

- fun wordCount list = mapReduce wc_mapper wc_reducer list;
val wordCount = fn : string list -> {count:int, word:string} list
```

Relational implementation of `mapReduce`

```
- fun mapReduce mapper reducer list =
=   from e in list,
=       (k, v) in mapper e
=     group k compute c = (fn vs => reducer (k, vs)) of v;
```

group …. compute

```
- fun median reals = ...;
val median = fn : real list -> real

- from e in emps
=   group x = e.deptno mod 3,
=         e.job
=   compute c = count,
=     sum of e.sal,
=     m = median of e.sal + e.comm;
val it = {c:int, job:string, m:real, sum:real, x.int} list
```

Join the discussion!

Constrained iterations, and Datalog-like recursion #106

julianhyde started this conversation in **Ideas**

julianhyde 5 days ago Maintainer edited

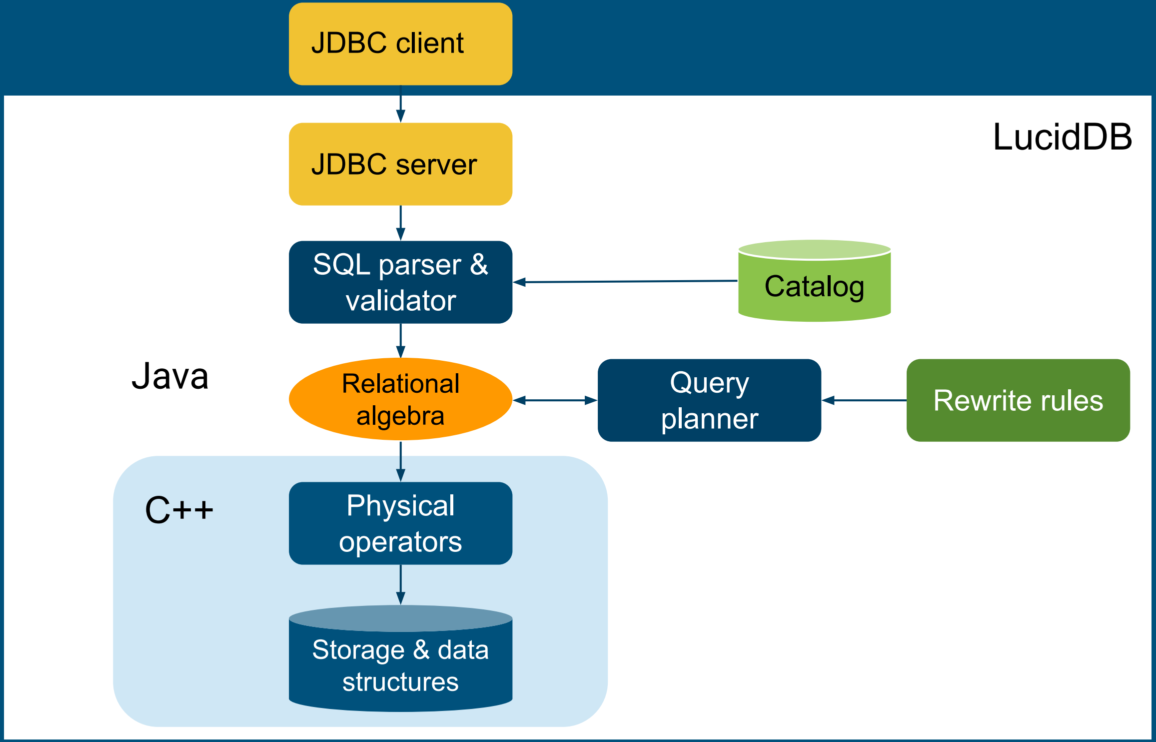
Morel has successfully integrated many aspects of relational languages into a functional programming language but a good syntax for recursively defined relations has been elusive. This is all the more surprising given that recursive relations are so easily defined in Datalog, a simpler language based on the deductive paradigm.

We propose a new Morel language concept, named 'constrained iteration' and introduced by a new `suchThat` keyword, that allows relations (including recursive relations) to be defined in a similar way to Datalog. This concept fits elegantly with the rest of Morel, gives Morel the same power as Datalog, and gives the Morel program authors more flexibility in how they structure their programs without sacrificing any power or efficiency.

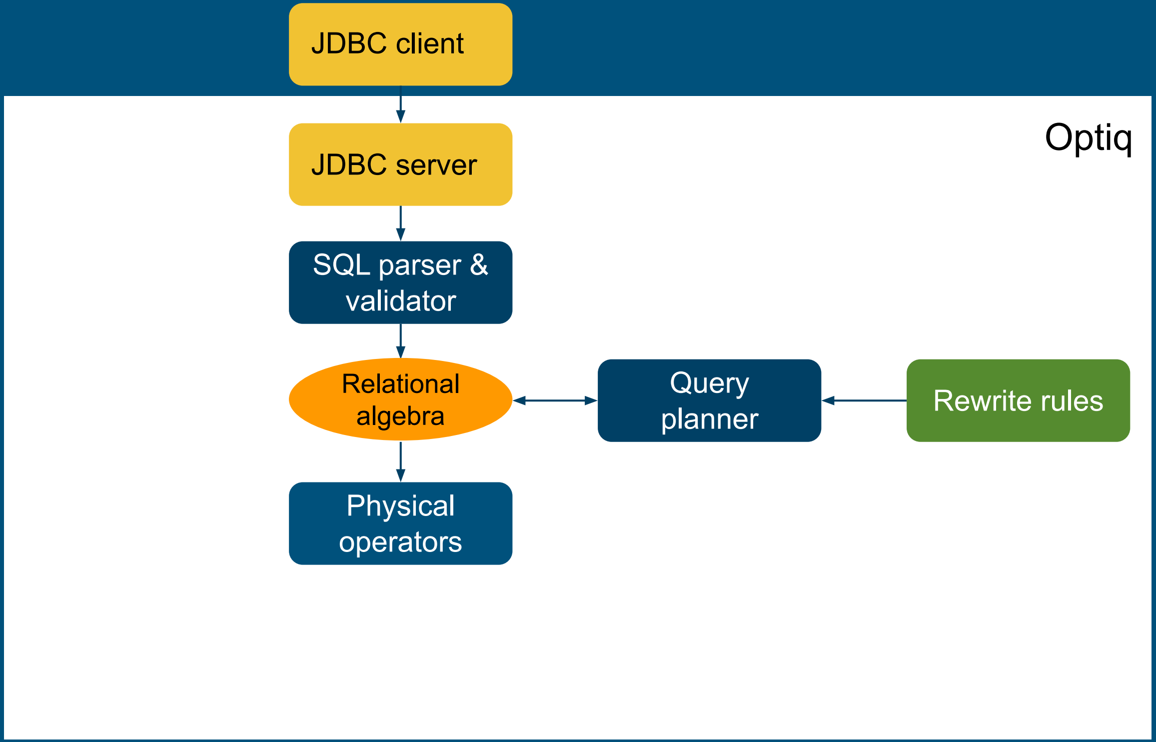
To solve the mystery of why Datalog is different, we journey through set theory, how mathematics and computer science define functions, and the relationship between fixed-points and recursively-defined functions.

https://github.com/hydromatic/morel/discussions/106

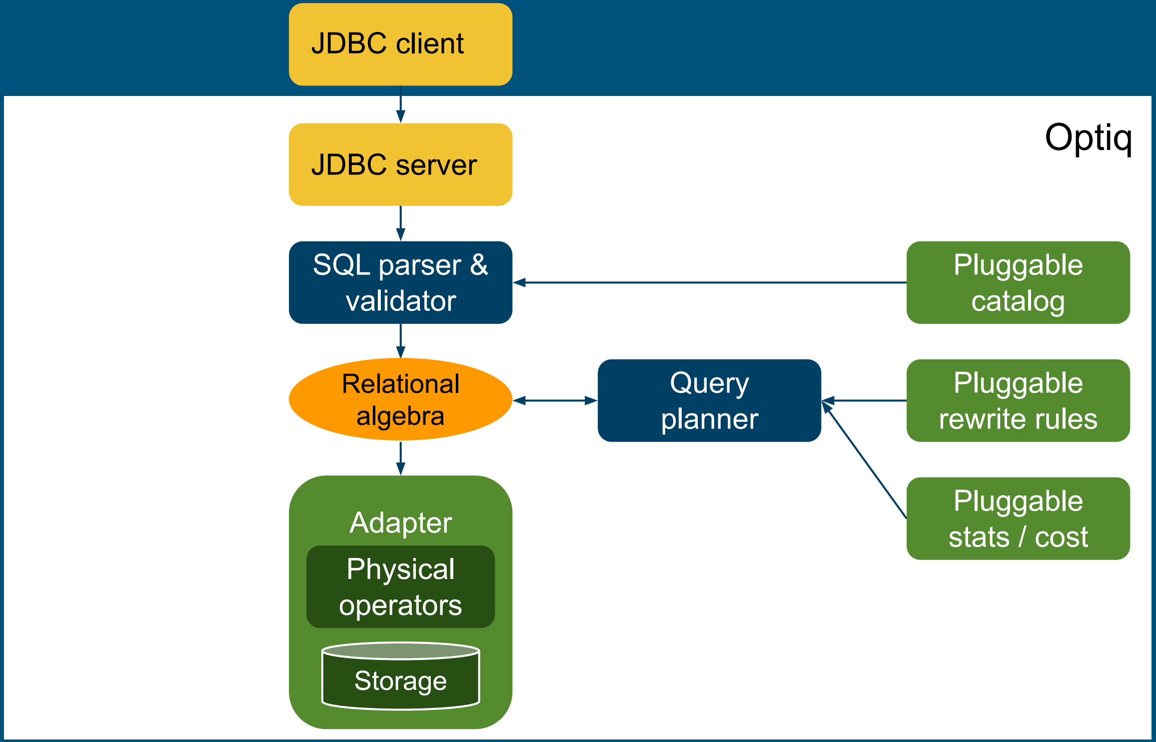
Calcite evolution - origins as an SMP DB



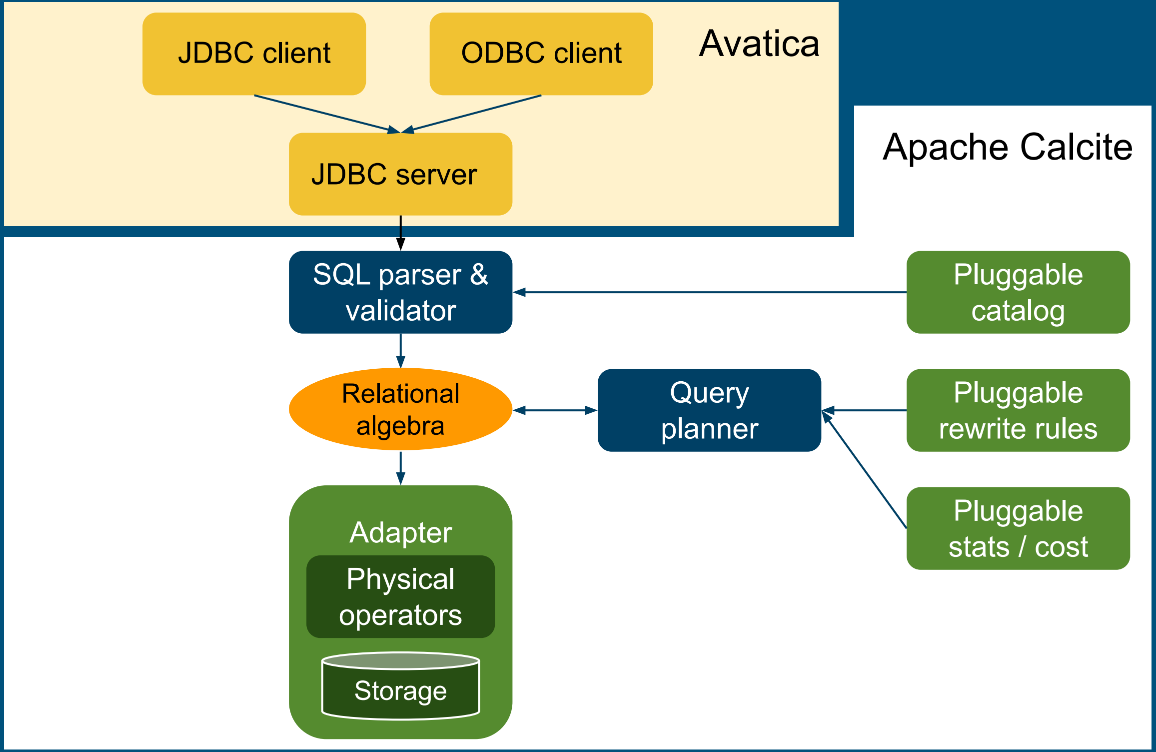
Calcite evolution - pluggable components



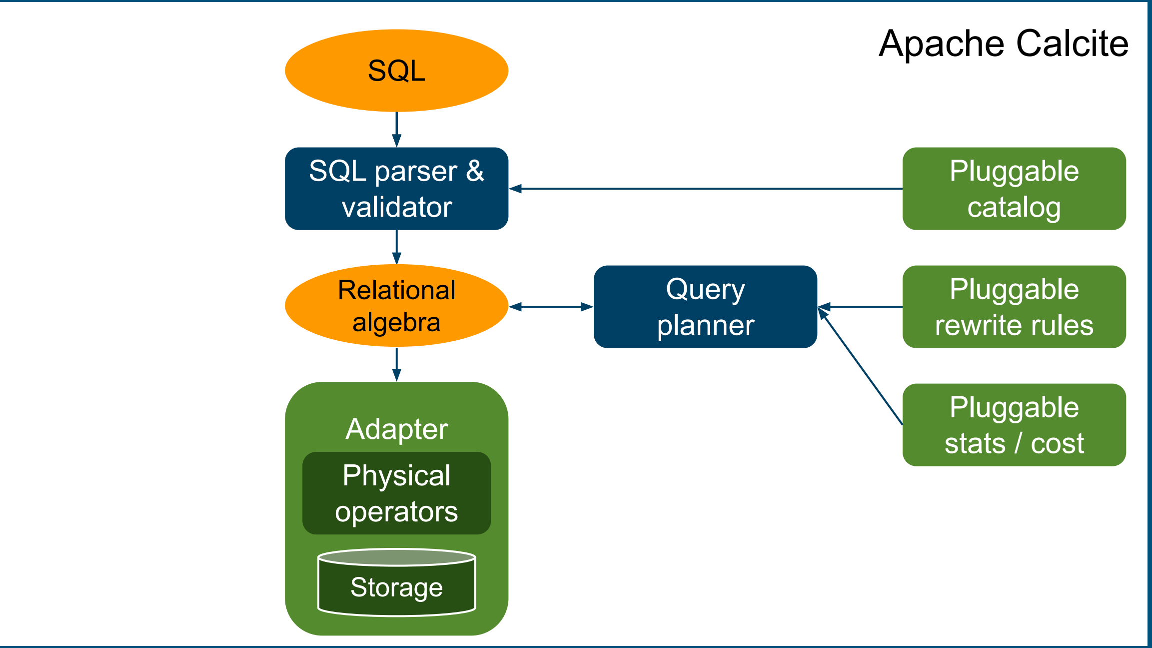
Calcite evolution - pluggable components



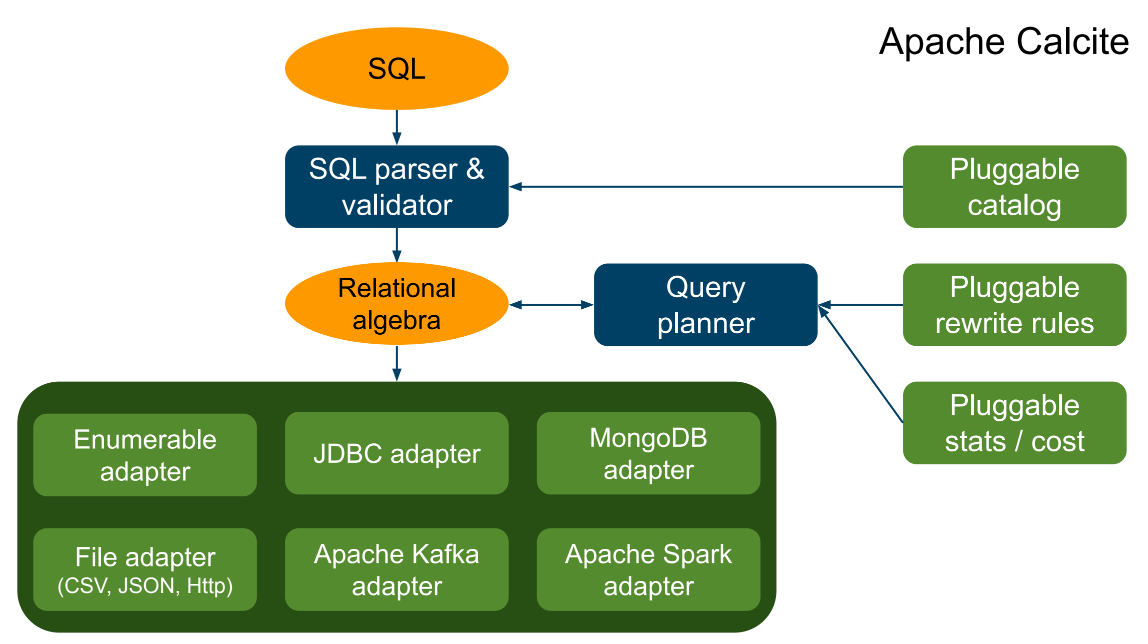
Calcite evolution - separate JDBC stack



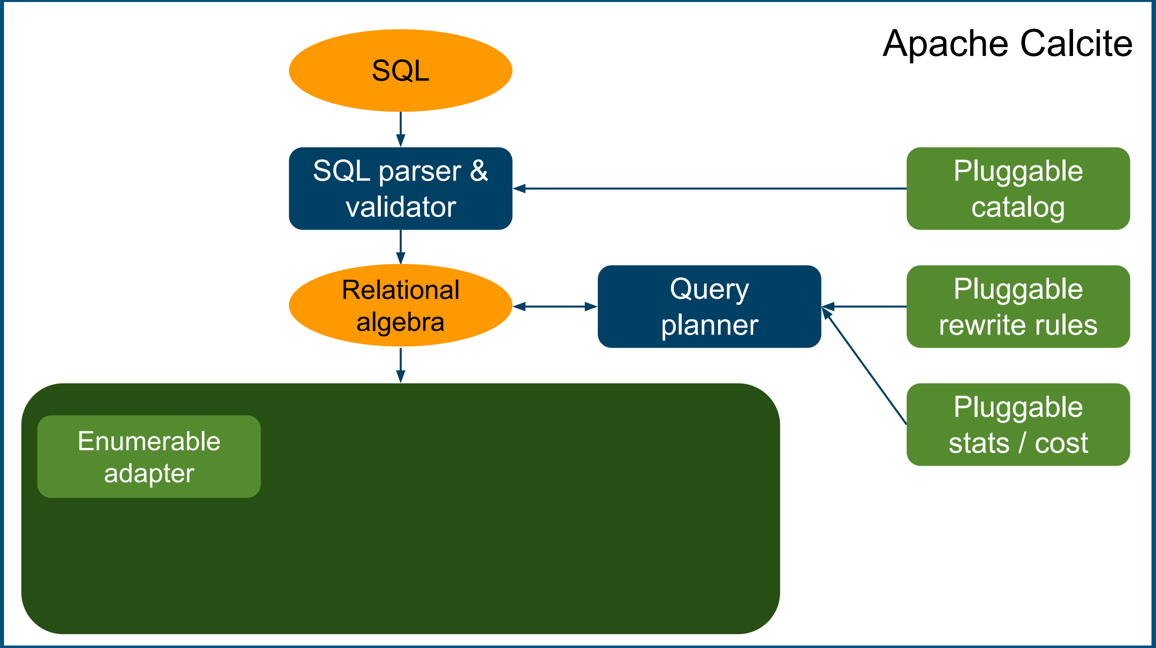
Calcite evolution - federation via adapters



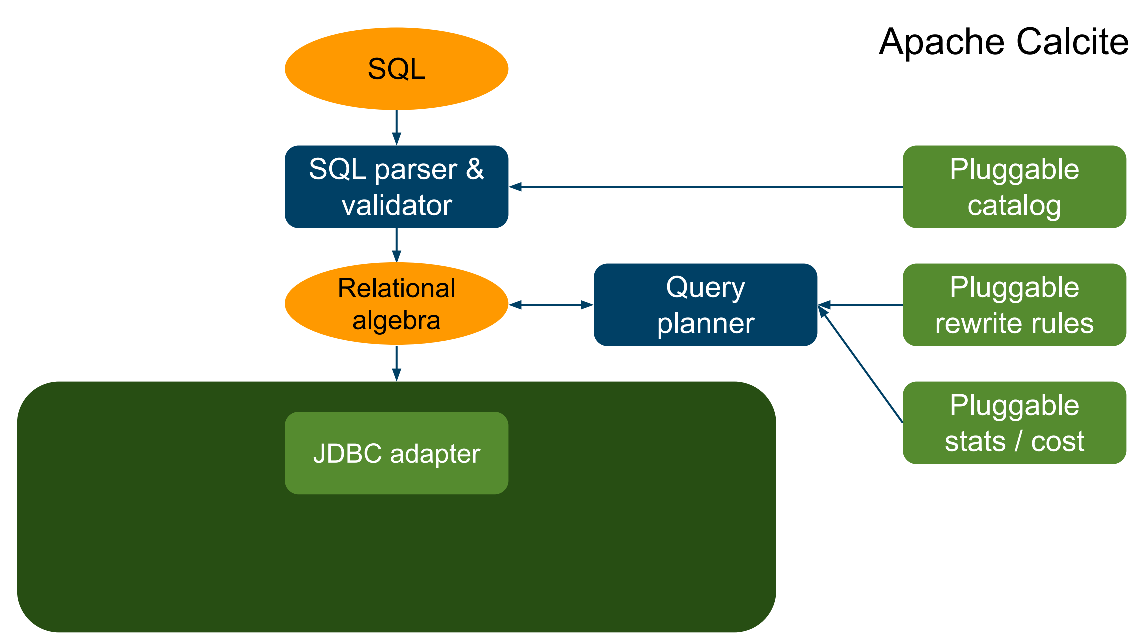
Calcite evolution - federation via adapters



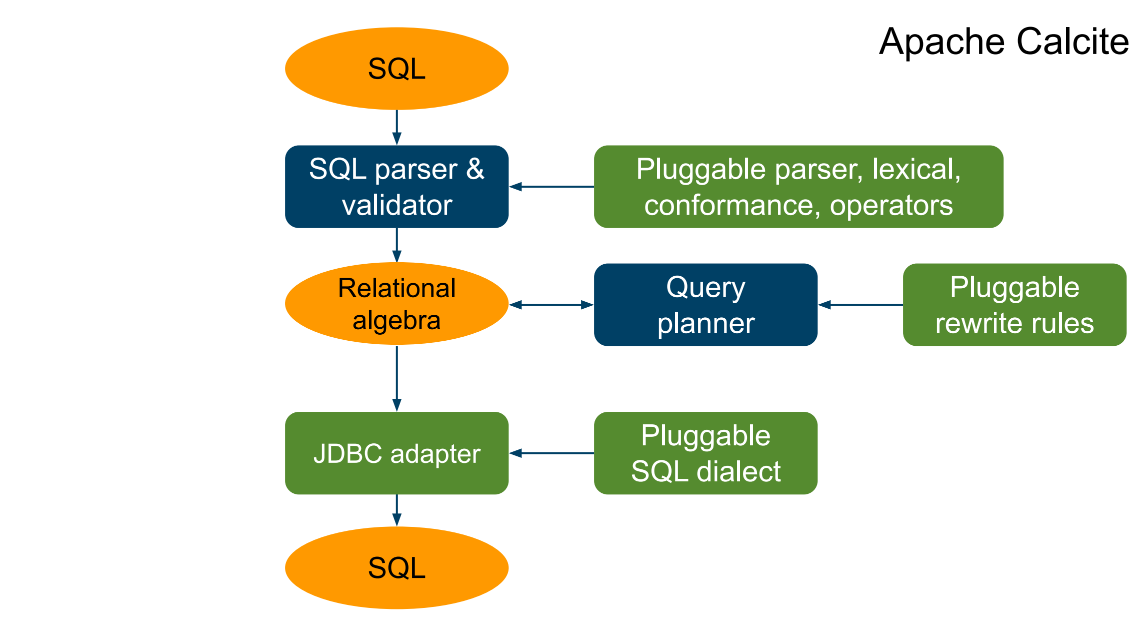
Calcite evolution - federation via adapters



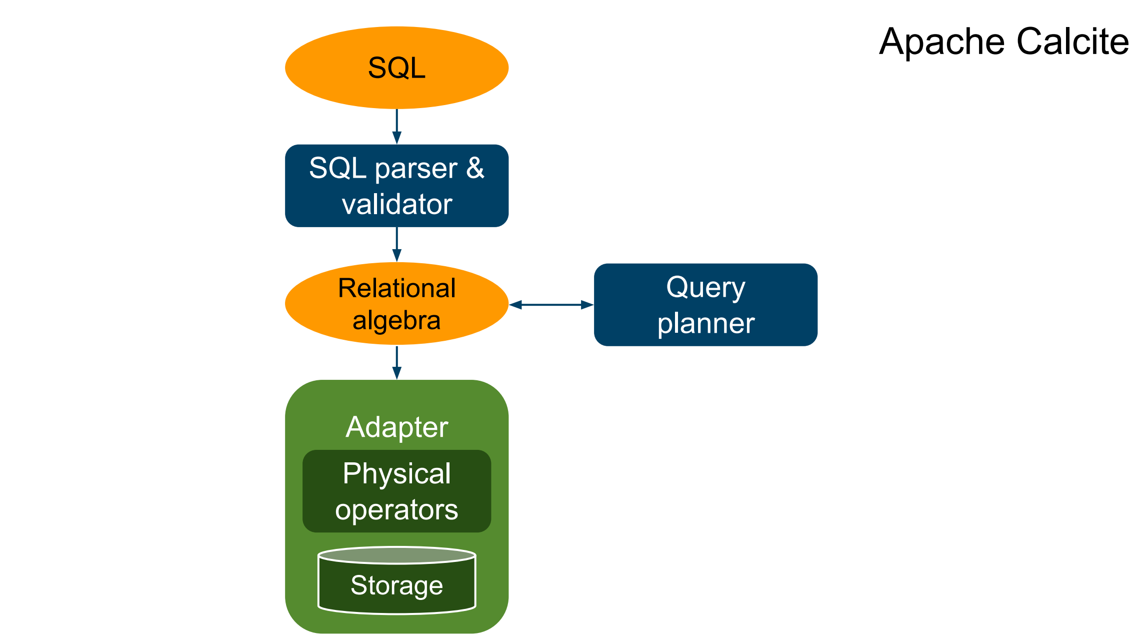
Calcite evolution - federation via adapters



Calcite evolution - SQL dialects



Calcite evolution - other front-end languages



Calcite evolution - other front-end languages