# More than Query

## Future Directions of Query Languages, from SQL to Morel

**Julian Hyde** (ex-Google; founder of Apache Calcite & Morel)

April 23rd, 2025 4:15 pm

Oakland

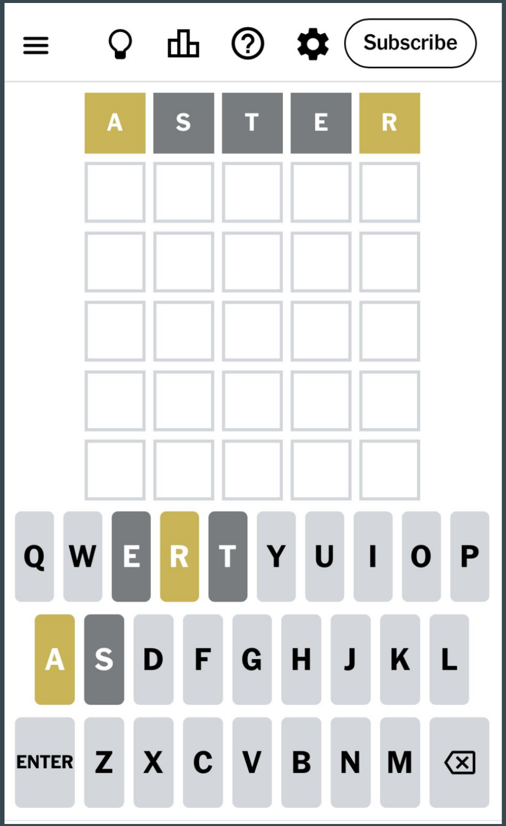
Data Council

# Solving Wordle

What is the best word to guess next?

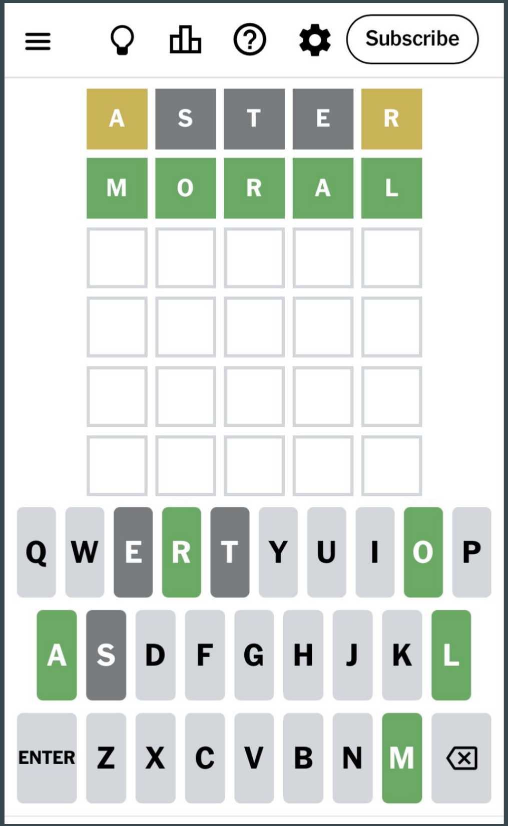
| Guess: WHARF | | Guess: MORAL | |
|---|---|---|---|
| W H A R F | CAIRN CARRY COBRA +57 | M O R A L | DRYAD RAJAH URBAN +4 |
| W H A R F | BOARD DIARY UNARY +19 | M O R A L | BURAN HYRAX YDRAD +10 |
| + 22 groups | | + 47 groups | |

For each possible guess,
    For each possible answer,
        Compute the mask for that guess for that answer
    Group answers that have the same mask
    Count the groups
Choose the guess that has the most groups



# Solving Wordle

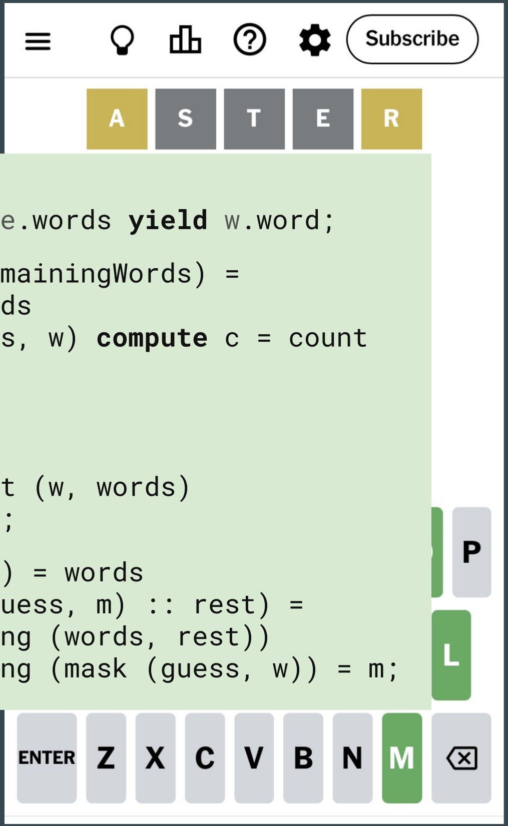
```
morel version 0.5.0 (java version "21.0.6", JRE null (build 21.0.6+8-LTS-188),
JLine terminal, xterm-256color)

- use "wordle.sml";
[opening wordle.sml]
val words =
  ["aahed","aalii","aargh","aarti","abaca","abaci","aback","abacs","abaft",
   "abaka","abamp","aband","abase","abash","abask","abate","abaya","abbas",
   "abbed","abbes",...] : string list
val mask = fn : string * string -> int
val maskToString = fn : int -> string
val maskCount = fn : string * string list -> int
val bestGuesses = fn : string list -> {maskCount:int, w:string} list
val remaining = fn : string list * (string * string) list -> string list
-
- bestGuesses (remaining (words, [("aster", "ybbby")]));
val it =
  [{maskCount=54,w="marid"},{maskCount=54,w="moria"},{maskCount=51,w="daric"},
   {maskCount=51,w="moira"},{maskCount=50,w="caird"},{maskCount=50,w="coria"},
   {maskCount=49,w="carbo"},{maskCount=49,w="cardi"},{maskCount=49,w="coram"},
   {maskCount=49,w="darcy"},{maskCount=49,w="moral"},{maskCount=48,w="maria"},
   {maskCount=47,w="baric"},{maskCount=47,w="borna"},{maskCount=47,w="cairn"},
   {maskCount=47,w="cardy"},{maskCount=47,w="carny"},{maskCount=47,w="caron"},
   {maskCount=47,w="cobra"},{maskCount=47,w="coral"},...]
  : {maskCount:int, w:string} list
```



# Solving Wordle

```
fun mask (guess, answer) =
  Let
    fun mask2 (m, i, [], answer) = m
      | mask2 (m, i, letter :: rest, answer) =
        mask2 ((m * 3
        + (if sub(answer, i) = letter
             then 2
           else if isSubstring(str letter) answer
             then 1
           else 0)), i + 1, rest, answer)
  in
    mask2 (0, 0, explode guess, answer)
  end;

fun maskToString m =
  let
    fun maskToString2 (m, s, 0) = s
      | maskToString2 (m, s, k) =
        maskToString2 (m div 3,
          List.nth(["b", "y", "g"], m mod 3) ^ s,
          k - 1)
  in
    maskToString2 (m, "", 5)
  end;
```

```
val words =
    from w in file.wordle.words yield w.word;

fun maskCount (guess, remainingWords) =
  from w in remainingWords
    group m = mask (guess, w) compute c = count
    compute count;

fun bestGuesses words =
  from w in words,
    maskCount = maskCount (w, words)
    order maskCount desc;

fun remaining (words, []) = words
  | remaining (words, (guess, m) :: rest) =
      from w in (remaining (words, rest))
        where maskToString (mask (guess, w)) = m;
```



# Agenda

What’s the difference between a query language and a programming language?

What’s wrong with SQL?

What things do other query-like languages do better?

Morel, a functional query language

# Algebra – a collection of values, operators, and laws

## On numbers

$a + (b + c) \rightarrow (a + b) + c$

$(a * b) + (a * c) \rightarrow a * (b + c)$

## On relations

$P \cup (Q \cup R) \rightarrow (P \cup Q) \cup R$

$(P \bowtie Q) \cup (P \bowtie R) \rightarrow P \bowtie (Q \cup R)$

# Functional programming – values, types, operators

```
1 + 2;
> val it = 3 : int

"Hello, " ^ "world!";
> val it = "Hello, world!" : string

val integers = [1, 2, 3, 4, 5, 6, 7, 8];
> val integers = [1,2,3,4,5,6,7,8] : int list

fun filter f [] = []
  | filter f (first :: rest) =
      if (f first)
        then first :: (filter f rest)
        else filter f rest;
val filter = fn : ('a -> bool) -> 'a list -> 'a list

filter (fn i => i mod 2 = 0) integers;
> val it = [2,4,6,8] : int list
```

# Relational algebra in a functional programming language

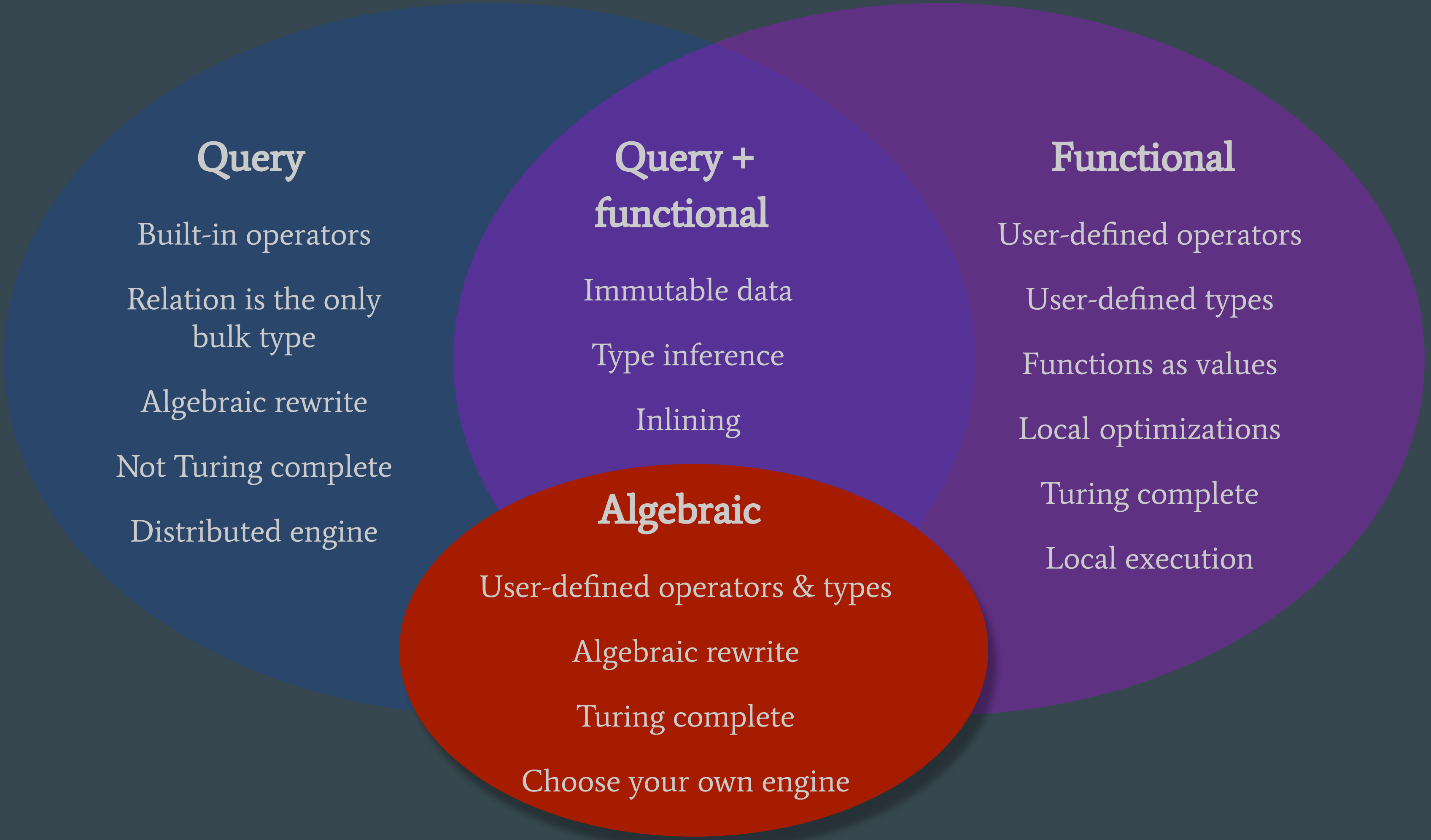
## Relational algebra

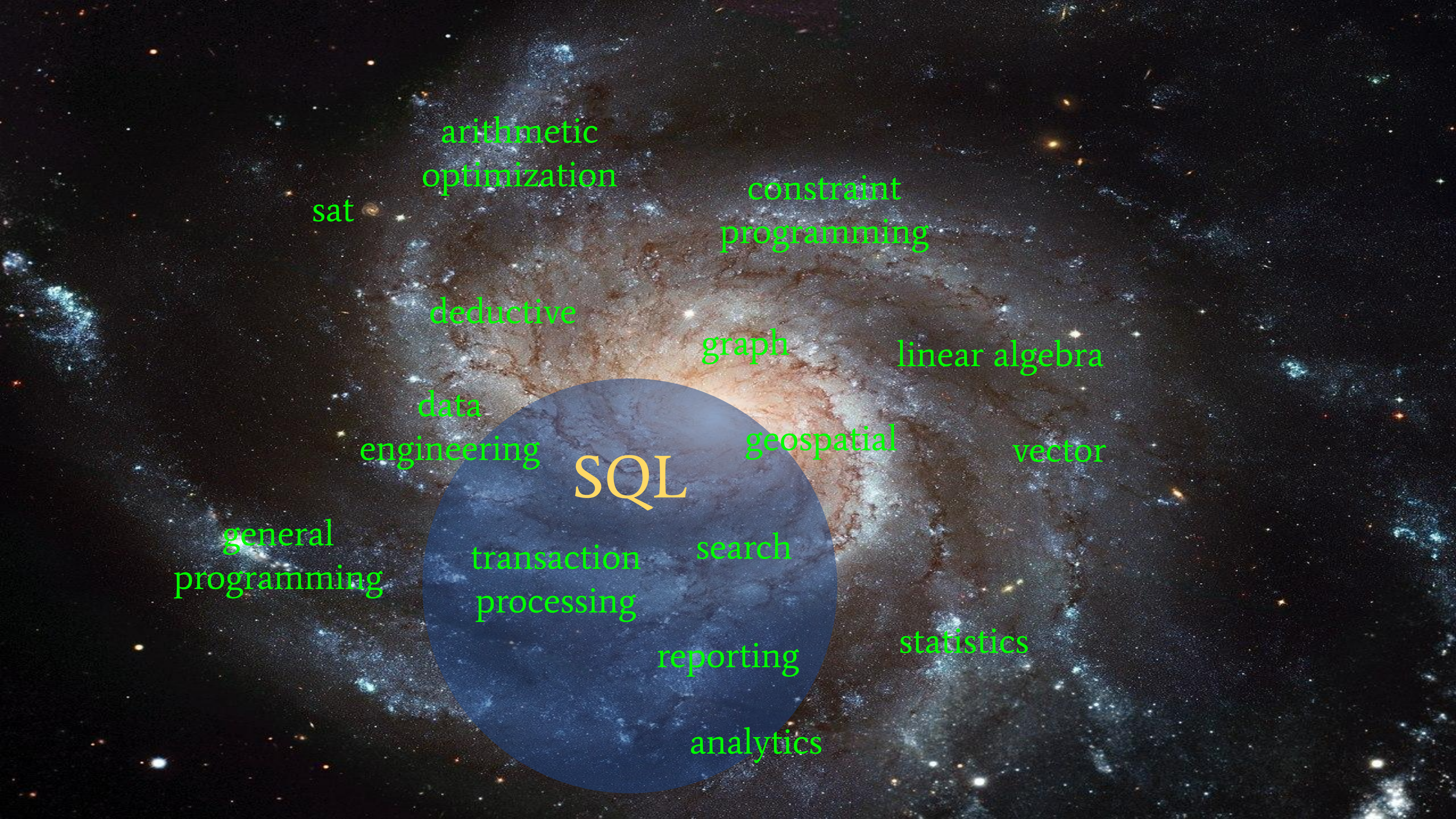
- $\cup$ union
- $\setminus$ minus
- $\cap$ intersect
- $\sigma$ filter
- $\Pi$ project
- $\bowtie$ join

## Relational operators as functions

```
val union = fn : 'a list * 'a list -> 'a list
val except = fn : 'a list * 'a list -> 'a list
val intersect = fn : 'a list * 'a list -> 'a list
val filter = fn : ('a -> bool) -> 'a list -> 'a list
val map = fn : ('a -> 'b) -> 'a list -> 'b list
val join = fn
  : 'a list * 'b list * ('a * 'b -> bool)
    -> ('a * 'b) list
```

# Is SQL a functional programming language?

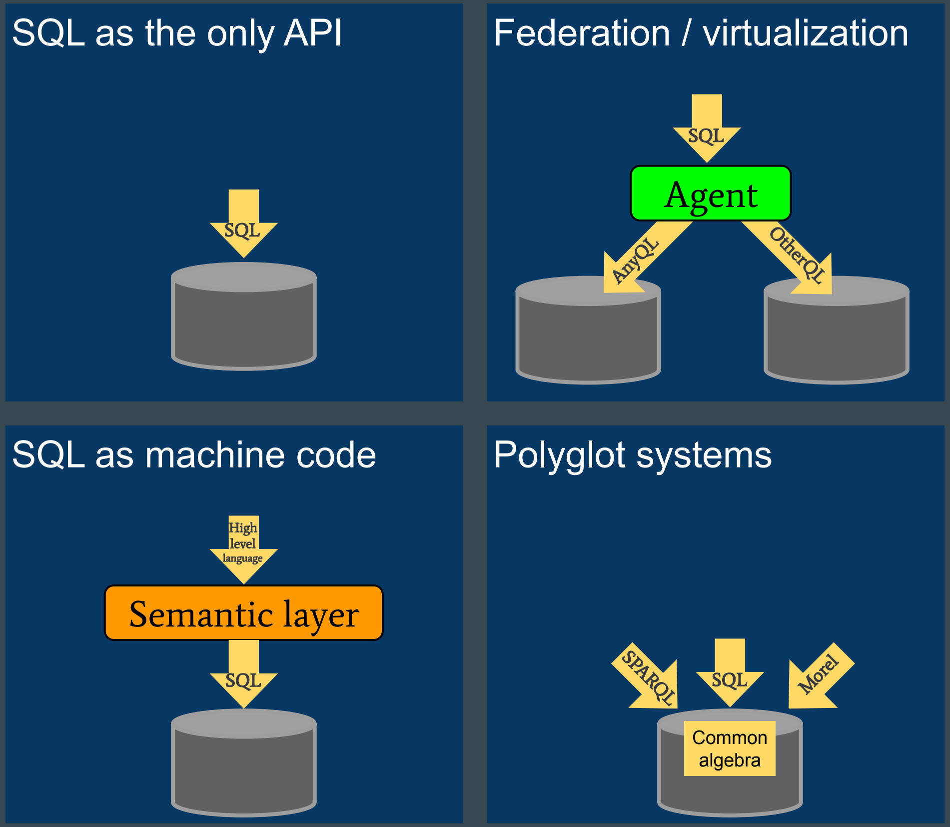

Query
Built-in operators
Relation is the only bulk type
Algebraic rewrite
Not Turing complete
Distributed engine
Query + functional
Immutable data
Type inference
Inlining
Functional
User-defined operators
User-defined types
Functions as values
Local optimizations
Turing complete
Local execution
Algebraic
User-defined operators & types
Algebraic rewrite
Turing complete
Choose your own engine


arithmetic
optimization
sat
constraint
programming
deductive
graph
linear algebra
data
engineering
geospatial
vector
SQL
general
programming
transaction
processing
search
statistics
reporting
analytics

# Future of SQL

I believe:

- SQL continues to grow/sprawl
- ISO standard declines in importance
- Databases speak more than one language
- Databases are not the only systems that speak SQL



# Alternatives to SQL

SQL

```sql
SELECT name, sal + comm AS pay
FROM emps
WHERE deptno = 10
```

MongoDB (JavaScript)

```javascript
db.emps.find(
  { deptno: 10 },
  { name: 1, pay: { $add: ["$sal", "$comm"] }, _id: 0 }
)
```

Datomic (Clojure)

```clojure
(d/q '[:find ?name ?pay
       :where
       [?e :emps/deptno 10]
       [?e :emps/name ?name]
       [?e :emps/sal ?sal]
       [?e :emps/comm ?comm]
       [(+ ?sal ?comm) ?pay]]
     db)
```

Spark (Scala)

```scala
df.filter($"deptno" === 10)
  .select($"name", ($"sal" + $"comm").alias("pay"))
```

Pandas (Python)

```python
df[(df['deptno'] == 10)][['name', 'sal', 'comm']] \
  .assign(pay=lambda x: x['sal'] + x['comm'])[['name', 'pay']]
```

Morel

```
from e in db.emps
  where e.deptno = 2
  yield {e.name, pay = e.sal + e.comm}
```

XQuery

```
for $emp in collection("emps")/employee
where $emp/deptno = 10
return
  <result>
    <name>{$emp/name/text()}</name>
    <pay>{$emp/sal/text() + $emp/comm/text()}</pay>
  </result>
```

LINQ (C#)

```
var result = from emp in emps
             where emp.deptno == 10
             select new {
                 name = emp.name,
                 pay = emp.sal + emp.comm
             };
```

## SQL

```
-- SQL
SELECT item, COUNT(*) AS c,
  SUM(sales) AS total
FROM ProduceSales
WHERE item != 'bananas'
  AND category IN ('fruit', 'nut')
GROUP BY item
ORDER BY item DESC;

item       c total
====== ==== =====
apples     2     9
```

## GoogleSQL pipe syntax

```
-- GoogleSQL pipe syntax
FROM ProduceSales
|> WHERE item != 'bananas'
    AND category IN ('fruit', 'nut')
|> AGGREGATE COUNT(*) AS c, SUM(sales) AS total
   GROUP BY item
|> ORDER BY item DESC;
```

```sql
-- SQL
SELECT item, COUNT(*) AS c,
  SUM(sales) AS total
FROM ProduceSales
WHERE item != 'bananas'
  AND category IN ('fruit', 'nut')
GROUP BY item
ORDER BY item DESC;

item       c total
====== ==== =====
apples     2     9
```

# SQL

# PRQL

```
# PRQL
from produceSales
filter item != "bananas"
filter category in ["fruit", "nut"]
group item (
  aggregate {
    c = count this,
    total = sum sales
  }
)
sort -item
```

# SQL

```sql
-- SQL
SELECT item, COUNT(*) AS c,
  SUM(sales) AS total
FROM ProduceSales
WHERE item != 'bananas'
  AND category IN ('fruit', 'nut')
GROUP BY item
ORDER BY item DESC;

item        c total
====== ==== =====
apples    2     9
```

Morel

```
(* Morel *)
from p in produceSales
  where p.item != "bananas"
    andalso p.category elem ["fruit", "nut"]
  group p.item compute c = count,
    total = sum of p.sales
  order item desc
```

# Morel query syntax

```
exp →
    from scan [, scan… ] [ step… ]
    [ terminalStep ]
  | exists scan [, scan… ] [ step… ]
  | forall scan [, scan… ] [ step… ]
    require condition
  | (* many other kinds of expression *)

scan →
    pat in collection [ on condition ]
  | pat = exp [ on condition ]
  | var

key → [ id = ] exp

agg → [ id = ] aggFn of exp

orderItem → exp [ desc ]
```

```
step →
    join scan [, scan… ]
  | where condition
  | yield exp
  | yieldmany collection
  | distinct
  | group key [, key… ]
    [ compute agg [, agg… ] ]
  | order item [, item… ]
  | skip number
  | take number
  | through pat in exp
  | union collection [, collection… ]
  | intersect collection [, collection… ]
  | except collection [, collection… ]

terminalStep →
    into exp
  | compute agg [, agg… ]
```

# The billion-dollar mistake

```sql
SELECT e.ename, e.sal
FROM emps AS e
WHERE e.deptno = 10
AND e.sal >
   (SELECT MAX(e2.sal)
    FROM emps AS e2
    WHERE e2.deptno = 20
    AND e2.job = 'PROGRAMMER')
```

# The billion-dollar mistake

Why do we need NULL?

- Empty strings
- Missing references
- Missing values
- N/A
- Unknown boolean
- Aggregation over the empty set

We don’t need NULL!

- Empty strings
- Option type
- Option type
- Option type
- Unnecessary
- Aggregate functions that return a default value, or an option

```sql
SELECT e.ename, e.sal
FROM emps AS e
WHERE e.deptno = 10
AND e.sal >
    (SELECT MAX(e2.sal)
     FROM emps AS e2
     WHERE e2.deptno = 20
     AND e2.job = 'PROGRAMMER')
```

```
from e in scott.emp
  where e.deptno = 10
    andalso
    (forall e2 in scott.emp
      where e2.deptno = 20
        andalso e2.job = "PROGRAMMER"
      require e.sal > e2.sal)
   yield e.ename
```

# Types and collections

## Atomic types

- **Primitive types** – e.g. `bool`, `char`, `string`, `int`, `real`
- **Abstract types** – e.g. `instant`, `interval`, `point`

## Collection types

- **Set** – no ordering, no duplicates
- **List** – ordering, duplicates allowed
- **Bag** – no ordering, duplicates allowed

A relation is a *set of records* (academia), a *bag of records* (industry)

## Functions and polymorphism

- **Type variables** – enable polymorphic types
- **Function types** – e.g. `int → bool`, `‘a → ‘b → ‘a list → ‘b list`

## Algebraic types

- **Record** – each value has several named fields
- **Sum type** – each value is one of several named variants

```
datatype personnel_id =
    EMPLOYEE of int
  | CONTRACTOR of {ssid: string, agency: string};

type member = {name: string, deptno: int, id: personnel_id};

val members = [
  {name = "Smith", deptno = 10, id = EMPLOYEE 100},
  {name = "Jones", deptno = 20,
   id = CONTRACTOR {ssid = "xxx-xx-xxxx", agency = "Cheap & cheerful"}];

val departments = scott.depts;

val primes = [2, 3, 5, 7, 11];

val bands = [["john", "paul", "george", "ringo"], ["simon", "garfunkel"]];
```

# Lists, bags, sets in Morel

Morel has `list` and `bag` types, but no `set` type. (`list` and `bag` have different operators; `set` is just `bag` plus constraints.)

Rules for a `from` expression:

- The output of a join or set operation is a `list` if all inputs are `list` values, otherwise a `set`;
- `order` keyword converts `bag` (or `list`) to `list`;
- `unorder` converts `list` (or `bag`) to `bag`;
- other operators (`where`, `group`, `distinct`, `yield`, `take`, `skip`) preserve collection type.

```
from i in [1, 2],
    j in ["a", "b"],
    k in [3, 4, 5, 6]
  where i + k < 6;
{i: int, j: string, k: int} list

from dept in [10, 30],
    e in scott.emps
  where e.deptno = dept
  yield e.ename;
string bag

from dept in [10, 30],
    e in scott.emps
  where e.deptno = dept
  order e.sal desc
  take 3
  yield {e.deptno, e.ename};
{deptno: int, ename: string} list
```

# Modifying data

# SQL DML

```sql
-- Delete employees who earn more than 1,000.
DELETE FROM scott.emps
WHERE sal > 1000;

-- Add one employee.
INSERT INTO scott.emps (empno, deptno, ename, job, sal)
VALUES (100, 20, 'HYDE', 'ANALYST', 1150);

-- Double the salary of all managers.
UPDATE scott.emps
SET sal = sal * 2
WHERE job = 'MANAGER';

-- Commit.
COMMIT;
```

# Morel DML (first attempt)

```
(* Delete employees who earn more than 1,000. *)
delete e in scott.emps
  where e.sal > 1000;

(* Add one employee. *)
insert scott.emps
  [{empno = 100, deptno = 20, ename = "HYDE",
    job = "ANALYST", sal = 1150}];

(* Double the salary of all managers. *)
update e in scott.emps
  where e.job = "MANAGER"
  assign (e, {e with sal = e.sal * 2});

(* Commit. *)
commit;
```

# Morel DML

```
(* Delete employees who earn more than 1,000. *)
val emps2 =
  from e in scott.emps
    where not (e.sal > 1000);

(* Add one employee. *)
val emps3 = emps2 union
  [{empno = 100, deptno = 20, ename = "HYDE", job = "ANALYST",
    sal = 1150}];

(* Double the salary of all managers. *)
val emps4 =
  from e in emps3
    yield if e.job = "MANAGER"
      then {e with sal = e.sal * 2}
      else e;

(* Commit. *)
commit {scott with emps = emps4};
```

# Incremental computation

```
(* New and removed employees. *)
val empsAdded = emps4 except scott.emps;
val empsRemoved = scott.emps except emps4;

(* Compute the updated summary table. *)
val summary2 =
  from s in scott.summary
    union
      (from e in empsAdded
        yield {e.deptno, c = 1, sum_sal = e.sal}
    union
      (from e in empsRemoved
        yield {e.deptno, c = ~1, sum_sal = ~sum_sal})
    group s.deptno compute c = sum of c, sum_sal = sum of sum_sal
    where c != 0);

(* Commit. *)
commit {scott with summary = summary2};
```

# Optimizing data-intensive programs

Query optimization (e.g. change join order, push down filters, parallelize)

Program optimization (e.g. remove unused variables, unwind loops, convert tail-recursion to loop)

Efficiently compute deltas

Recommend materialized views

Add temporary tables for shared intermediate results

Remove unused temporary tables

Multi-query optimization [CALCITE-6188]

# Deductive query,
# Graph query, and
# Optimization problems

# Parents (a base relation)

```
-- SQL
SELECT *
FROM parents;

parent   child
======== =======
earendil elrond
elrond   arwen
elrond   elladan
elrond   elrohir

SELECT *
FROM parents
WHERE parent = 'elrond';

parent child
====== =======
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
```

```
(* Morel *)
from (parent, child) in parents
  where parent = "elrond";
[("elrond", "arwen"),
 ("elrond", "elladan"),
 ("elrond", "elrohir")]
```

# Ancestors (a recursively-defined relation)

```
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
```

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

```
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).
X = elrond
X = earendil
```

```
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;
["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,
    d in depts
  where isClerk(e)
    andalso d.deptno = e.deptno
    andalso d.loc = "DALLAS"
  yield e.name;
["SMITH", "ADAMS"] : string list;
```

Unbounded variable **e** iterates over all values that could yield a result.

# Recursively-defined predicate relation

```
(* Morel *)

fun isAncestor (x, z) =
  (x, z) elem parents
    orelse (exists y
      where isAncestor(x, y)
        andalso (y, z) elem parents);

from a
  where isAncestor(a, "arwen");
[“earendil”, “elrond”] : string list

from d
  where isAncestor("earendil", d);
[“elrond”, “arwen”, “elladan”, “elrohir”] : string list
```

# Recipe optimization

**Available ingredients**

- 40kg flour
- 60 bananas
- 20kg sugar
- 5kg butter
- 5kg cocoa

**Banana cake $4**

- 50g flour
- 2 bananas
- 75g sugar
- 100g butter

**Chocolate cake $4.50**

- 200g flour
- 150g sugar
- 150g sugar
- 75g cocoa

```
from b, c
  where b >= 0 andalso b <= 100          (*) number of banana cakes
  andalso c >= 0 andalso c <= 100        (*) number of chocolate cakes
  andalso 50 * b + 200 * c <= 40000      (*) flour
  andalso 2 * b <= 60                    (*) bananas
  andalso 75 * b + 150 * c <= 20000      (*) sugar
  andalso 100 * b + 150 * c <= 5000      (*) butter
  andalso 75 * c <= 5000                 (*) cocoa
  yield {b, c, profit = 400 * b + 450 * c}
  order profit desc
  take 3;
val it =
  [{b=29,c=14,profit=17900},
   {b=30,c=13,profit=17850},
   {b=26,c=16,profit=17600}] : {b:int, c:int, profit:int} list
```

If only... using a database could be more like using a programming language

# Many nice things about programming languages

If it compiles, it probably works

Refactoring

Autocompletion

Git

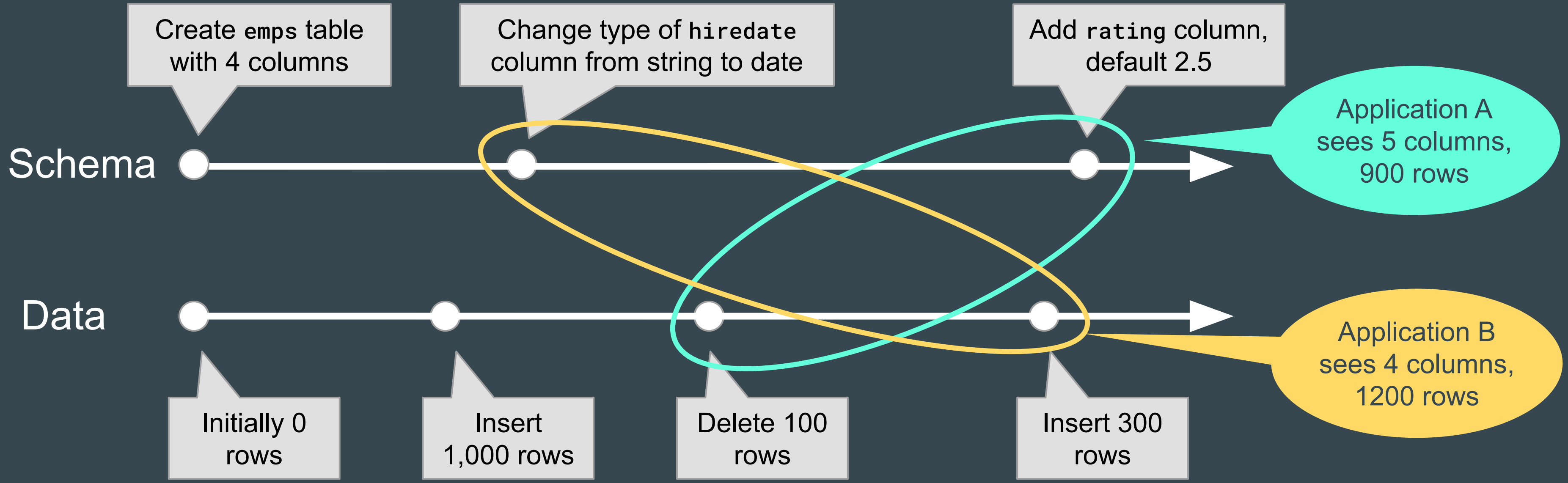
Documentation in the code

Unit tests in the same language

Modules

Abstraction

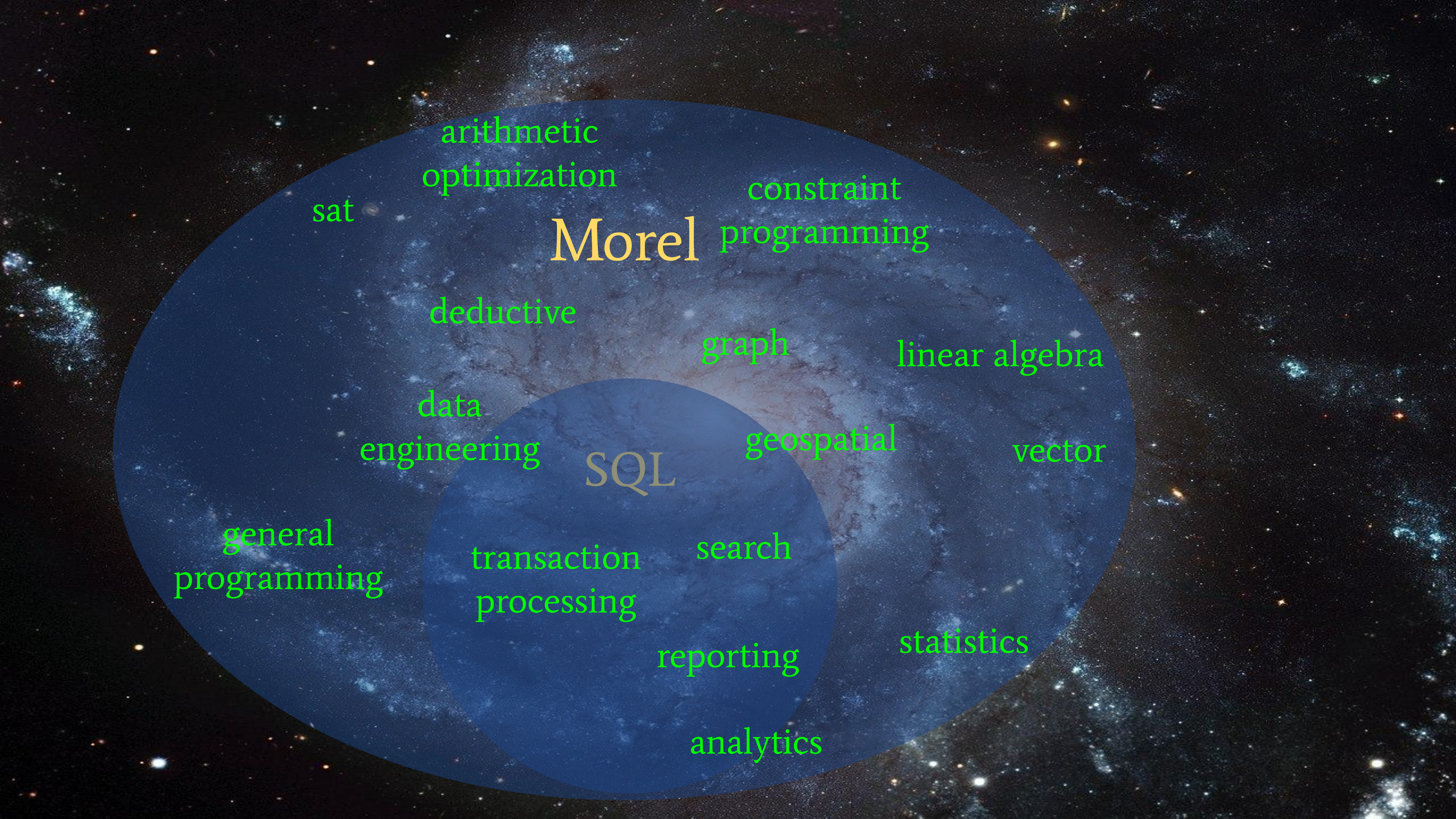
# Types and data evolve independently



# Database as a module (shims for schema evolution)

```
(* Initial database value and type (schema). *)
val scott1 = db
  : {emps: {name: string, empno: int, deptno: int,
            hiredate: string} bag,
    depts: {deptno: int, name: string} bag};

(* Shim that makes a v1 database look like v2. *)
fun scott2on1shim scott1 =
  {emps =
    fn () => from e in scott1.emps
      yield {e with hiredate = Date.fromString(e.hiredate)},
   depts = fn () => scott1.depts};

(* Shim that makes v3 database look like v1. *)
fun scott1on3shim scott3 =
  {emps =
    fn () => from e in scott3.emps
      yield {e with hiredate = Date.toString(e.hiredate)
                removing rating},
   depts = fn () => scott3.depts};

(* An application writes its queries & views against version 2;
   shims make it work on any actual version. *)
val scott = scott2;
fun recentHires () =
  from e in scott.emps
    where e.hiredate > Date.subtract(Date.now(), 100);
```

# Conclusions


arithmetic
optimization
sat
Morel
constraint
programming
deductive
graph
linear algebra
data
engineering
SQL
geospatial
vector
general
programming
transaction
processing
search
reporting
statistics
analytics

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

Yes, we can do better than SQL!
Best of both query languages + programming languages
Give Morel a try!