More than query

Morel, SQL, and the evolution of data languages

Julian Hyde

October 29th-30th, 2025

Current 2025, New Orleans

More than query: Morel, SQL and the evolution of data languages

Wednesday, October 29, 2025
2:45 PM - 3:30 PM

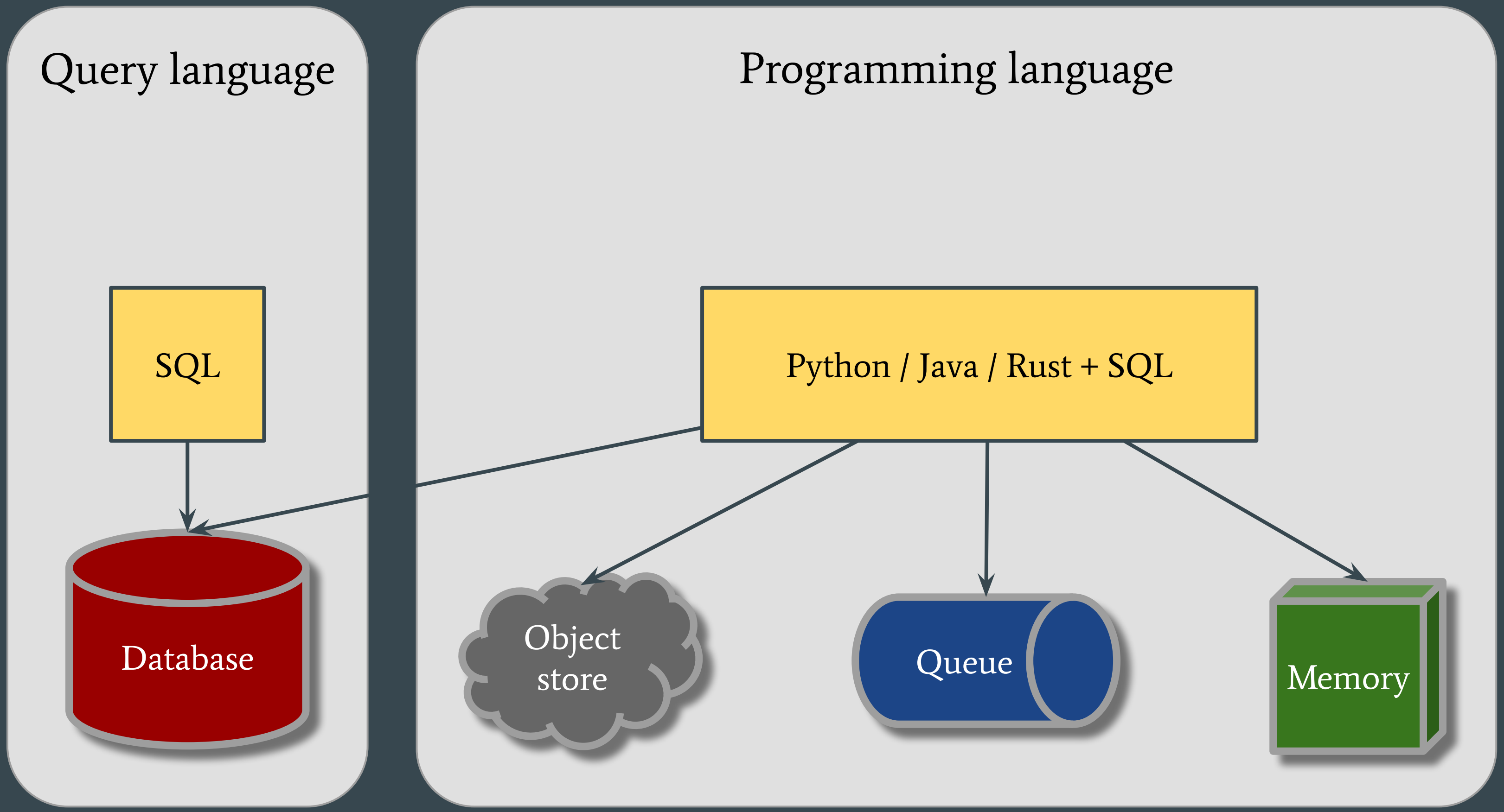
What is the difference between a query language and a general-purpose programming language? Can SQL be extended to support streaming, incremental computation, data engineering, and general-purpose programming? How well does SQL fit into a modern software engineering workflow, with Git-based version control, CI, refactoring, and AI-assisted coding?

These are all questions that drove the creation of Morel. Morel is a new functional programming language that embeds relational algebra, so it is as powerful as SQL. Morel's compiler, like that of any good SQL planner, generates scalable distributed programs, including federated SQL. But unlike SQL, Morel is Turing-complete, which means that you can solve the whole problem without leaving Morel.

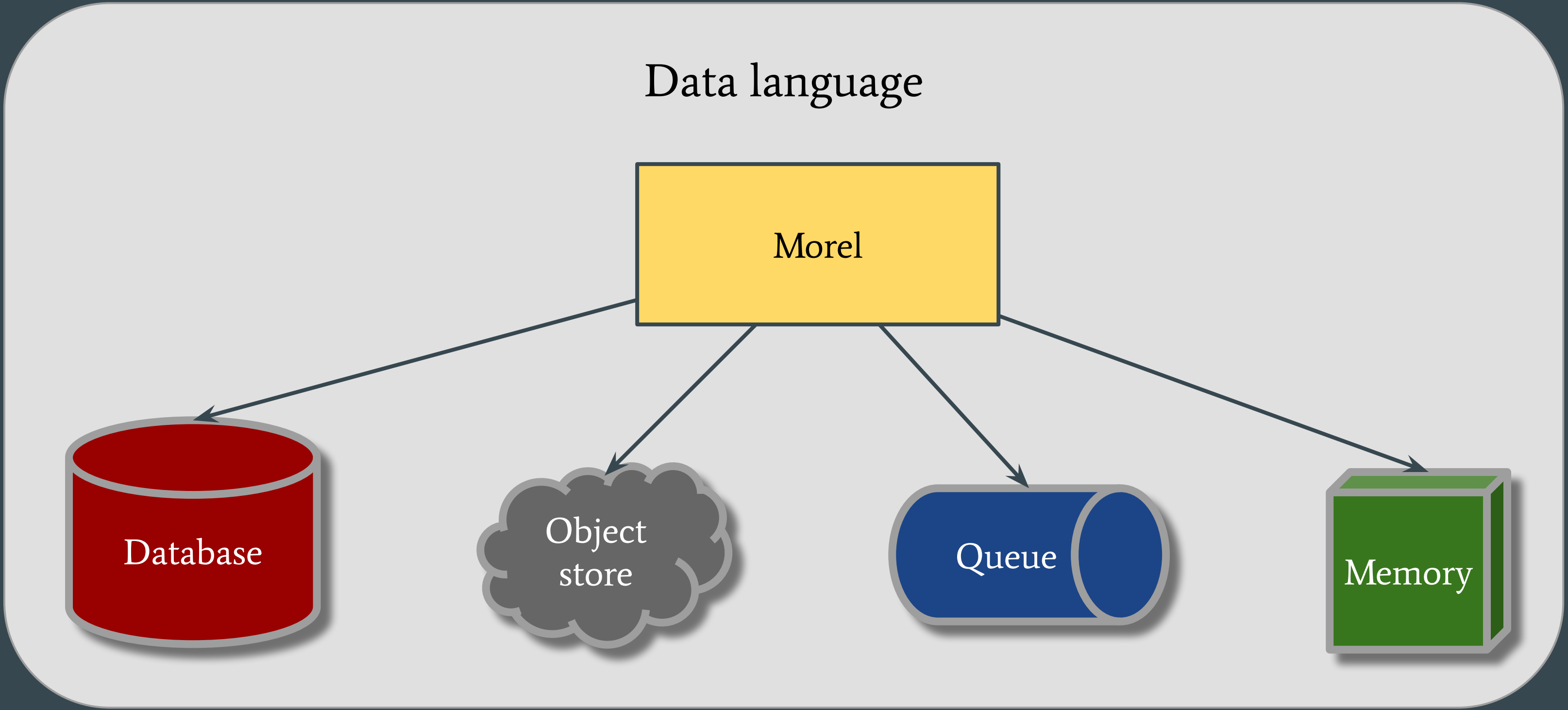
This session will discuss the challenges and opportunities of query languages, especially for streaming and data engineering tasks, and provide a gentle introduction to the Morel language. It is presented by Morel's creator, Julian Hyde, who created Apache Calcite and also pioneered streaming SQL.

Location	Breakout Room 285
Level	Intermediate
Audience	Data Engineer/Scientist, Data Streaming Engineers
Track	Storage and Analytics

Generated by Google Gemini



Query language
Programming language
SQL
Python / Java / Rust + SQL
Database
Object store
Queue
Memory


Data language
Morel
Database
Object store
Queue
Memory

About me

SQLstream
Query the Future™

APACHE
calcite™

kafka summit
Real-time data
Stream processing
Massive scale
SF 4.26.2016

Looker

Morel

Agenda

1. Data language
2. High level language
3. SQL
4. A comfortable programming language
5. Better query

1. Data language

What do we want from a data language?

1. Data

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

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

(3.14, true);
> val it = (3.14, true) : real * bool

{empno = 100, ename = "SCOTT", job = "MANAGER"};
> val it : {empno:int,ename:string,job:string}

val depts = [
  {deptno = 10, dname = "SALES", emps = []},
  {deptno = 20, dname = "MARKETING", emps = [
    {empno = 100, ename = "SCOTT", job = "MANAGER"},
    {empno = 110, ename = "OATES", job = "CLERK"}
  ]}];
> val depts
  : {deptno:int,dname:string,
      emps:{empno:int, ename:string, job:string} list} list
```

What do we want from a data language?

1. Data
2. **Functions**

```
fn x => x * x;
> val it = fn : int -> int

map (fn x => x * x) [1, 2, 3, 4];
> val it = [1,4,9,16] : int list

fun factorial 1 = 1
  | factorial n = n * (factorial (n - 1));
> val factorial = fn : int -> int
```

What do we want from a data language?

1. Data
2. Functions
3. **Types**

```
type employee =
  {empno:int, ename:string, is_mgr:bool,
   mgrno:int option};
> type employee

datatype color = BLUE | GREEN | RED;
> datatype color = BLUE | GREEN | RED

datatype ‘a option = NONE | SOME of ‘a;
> datatype option

SOME "abc";
> val it = SOME “abc” : string option

NONE;
> val it = NONE : ‘a option
```

What do we want from a data language?

1. Data
2. Functions
3. Types
4. **Constraints**

```
type nat = int check (fn v => v >= 0);
> type nat

type empno = nat;
> type empno

type hr = {
  emps: employee bag check (fn emps =>
    not (exists e in emps
      group e.empno compute count
      where count > 1),
  depts: department bag check (fn depts =>
    not (exists d in depts
      group d.deptno compute count
      where count > 1)
} check (fn hr =>
    not (exists e in hr.emps yield e.deptno
      except distinct
      (from d in hr.depts yield d.deptno)));
> type hr
```

What do we want from a data language?

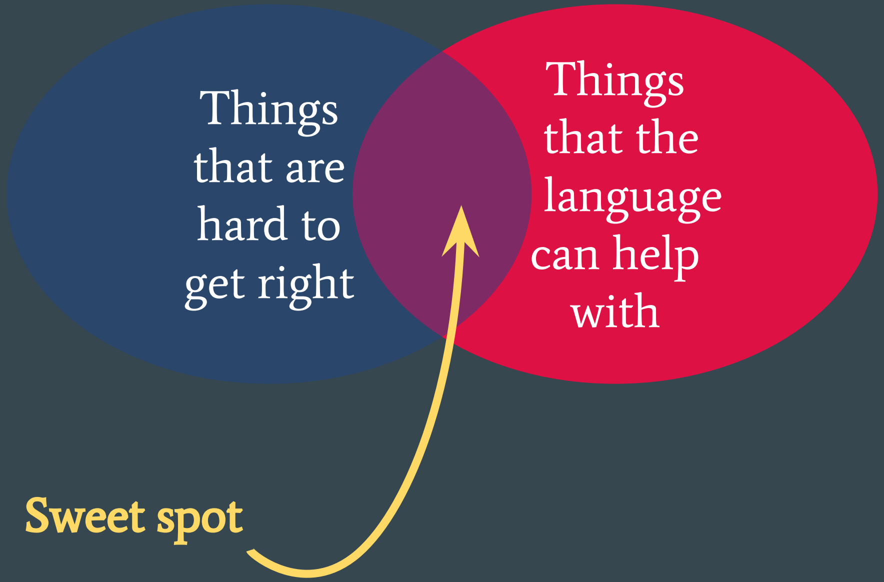
1. Data
2. Functions
3. Types
4. Constraints
5. **Expressions**

```
substring ("abcde", 1, 2);
> val it = “bc”: string

tl [1, 2, 3];
> [2,3] : list

from i in [1, 2, 3, 4, 5]
 where i mod 2 = 1
 yield i * i;
> [1,9,25]: int list

fun categorize (x, y) =
  case (x mod 2, y mod 2) of
      (0, 0) => "both even"
    | (1, 1) => "both odd"
    | (_, _) => "odd and even";
> val categorize = fn : int * int -> string
```

2. High level language

The “sweet spot” of language design

1. Organization of data
2. Choice of algorithms
3. Parallelism/federation
4. Mutation
5. Evolving schema
6. Abstraction



high-level programming language *n*.
A programming language that requires you to specify only the details that matter.

morel *n*.

1. An edible mushroom with a honeycomb-like cap.
2. (*comp. sci*.) A high-level data programming language that combines Standard ML with relational algebra.

What’s a high-level language?

Part 1: Java ArrayList

```
record LogEntry(String timestamp, String message) {}

List<LogEntry> logsList = List.of(
    new LogEntry("2025-10-25T08:15:23", "User login: alice"),
    new LogEntry("2025-10-25T08:16:45", "API call: /users"),
    new LogEntry("2025-10-25T09:23:11", "Error: timeout"),
    new LogEntry("2025-10-25T10:05:33", "User login: bob"),
    new LogEntry("2025-10-25T14:22:01", "API call: /orders")
    // ... millions of log entries
    );

List<String> logsInRange(String startTime,
   String endTime, List<LogEntry> logs) {
  var result = new ArrayList<String>();
  for (var entry : logs) {
    if (entry.timestamp.compareTo(startTime) >= 0
        && entry.timestamp.compareTo(endTime) <= 0) {
      result.add(entry.message);
    }
  }
  return result;
}

logsInRange("2025-10-2509:00:00", "2025-10-2511:00:00", logsList);
```

What’s a high-level language?

Part 2: Java SortedMap

```java
SortedMap<String, String> logsMap =
    new TreeMap<>(
        Map.ofEntries(
            Map.entry("2025-10-25T08:15:23", "User login: alice"),
            Map.entry("2025-10-25T08:16:45", "API call: /users"),
            Map.entry("2025-10-25T09:23:11", "Error: timeout"),
            Map.entry("2025-10-25T10:05:33", "User login: bob"),
            Map.entry("2025-10-25T14:22:01", "API call: /orders")
            // ...
        ));

List<String> logsInRange(String startTime,
    String endTime, SortedMap<String, String> logs) {
  var result = new ArrayList<String>();
  var subMap = logs.subMap(startTime, true, endTime, true);
  for (var message : subMap.values()) {
    result.add(message);
  }
  return result;
}

logsInRange("2025-10-2509:00:00", "2025-10-2511:00:00", logsMap);
```

What’s a high-level language?

Part 3: Rust BTreeMap

```
let mut logs_map = BTreeMap::new();
logs_map.insert("2025-10-25T08:15:23".to_string(),
    "User login: alice".to_string());
logs_map.insert("2025-10-25T08:16:45".to_string(),
    "API call: /users".to_string());
logs_map.insert("2025-10-25T09:23:11".to_string(),
    "Error: timeout".to_string());
logs_map.insert("2025-10-25T10:05:33".to_string(),
    "User login: bob".to_string());
logs_map.insert("2025-10-25T14:22:01".to_string(),
    "API call: /orders".to_string());

fn logs_in_range(start_time: &str, end_time: &str,
    logs: &BTreeMap<String, String>) -> Vec<String> {
    let mut result = Vec::new();
    for (_timestamp, message)
      in logs.range(start_time.to_string()..=end_time.to_string()) {
        result.push(message.clone());
    }
  result
}

logs_in_range("2025-10-2509:00:00", "2025-10-2511:00:00", logsList);
```

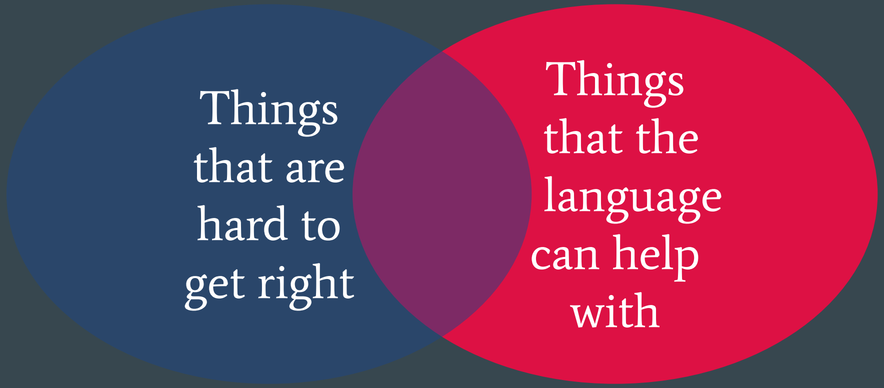
What's a high-level language?

Part 4: Morel list

```
val logsList = [
  ("2025-10-25T08:15:23", "User login: alice"),
  ("2025-10-25T08:16:45", "API call: /users"),
  ("2025-10-25T09:23:11", "Error: timeout"),
  ("2025-10-25T10:05:33", "User login: bob"),
  ("2025-10-25T14:22:01", "API call: /orders")
  (* ... millions of log entries *)
];

fun logsInRange (startTime, endTime, logs) =
  from (timestamp, message) in logs
    where timestamp >= startTime
      andalso timestamp <= endTime
    yield message;

logsInRange ("2025-10-2509:00:00", "2025-10-2511:00:00", logsList);
```

The “sweet spot” of language design

1. Organization of data
2. Choice of algorithms
3. Parallelism/federation
4. Mutation
5. Evolving schema
6. Abstraction



3. SQL

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

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

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

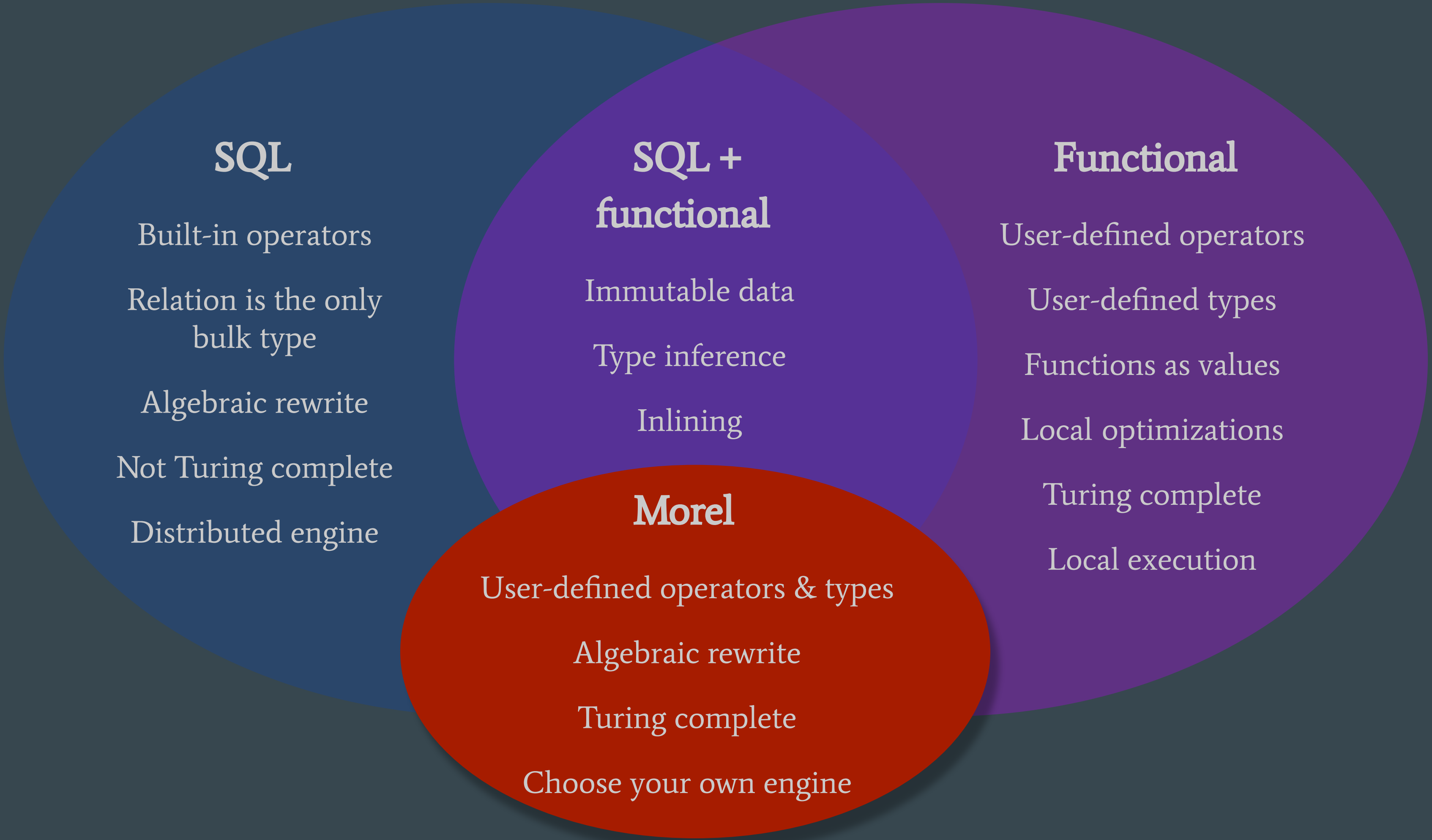
Relational algebra in a functional programming language

Relational algebra

- ∪ union
- \ minus
- ∩ intersect
- σ filter
- Π project
- ⋈ 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
```

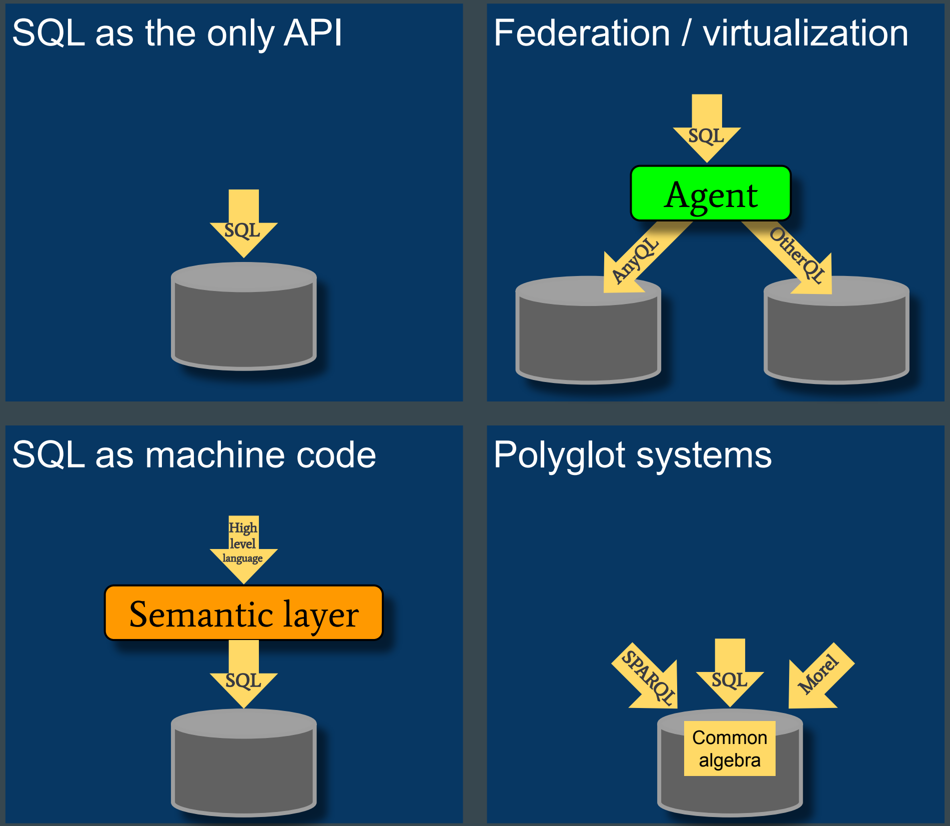

SQL
Built-in operators
Relation is the only bulk type
Algebraic rewrite
Not Turing complete
Distributed engine
SQL + functional
Immutable data
Type inference
Inlining
Functional
User-defined operators
User-defined types
Functions as values
Local optimizations
Turing complete
Local execution
Morel
User-defined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

Future of SQL

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



Data systems

Data systems

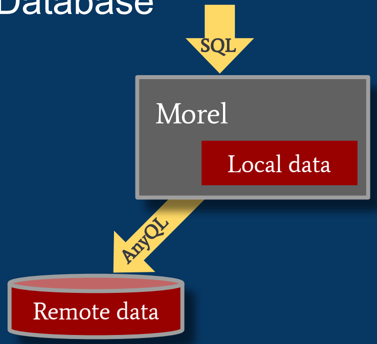
Command-line executable

```
% morelc my_prog
% target/my_prog arg
```

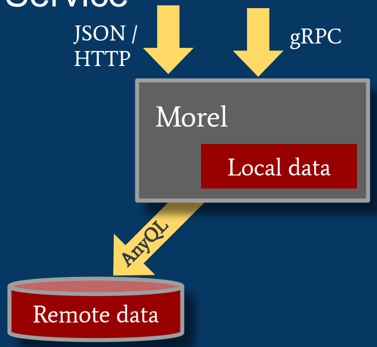
Shell

```
> from e in emps
  where e.mgr
  compute count;
val it = 17 : int
```

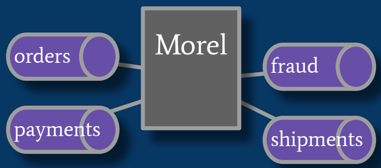
Database



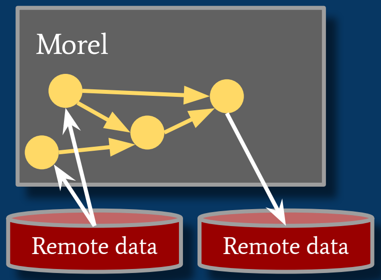
Service



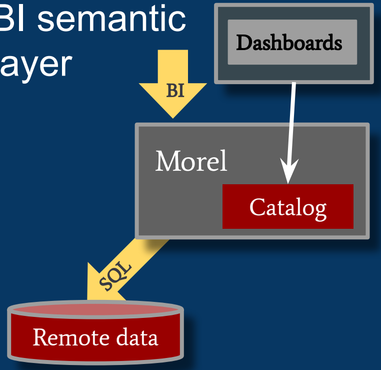
Event processor



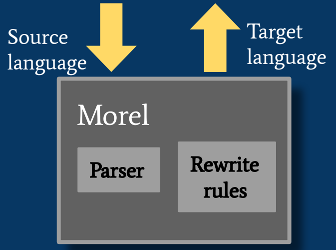
ETL / orchestration



BI semantic layer



Translator



4. A comfortable programming language for your data

Many nice things about programming languages

If it compiles, it probably works

Refactoring

Autocompletion

Git

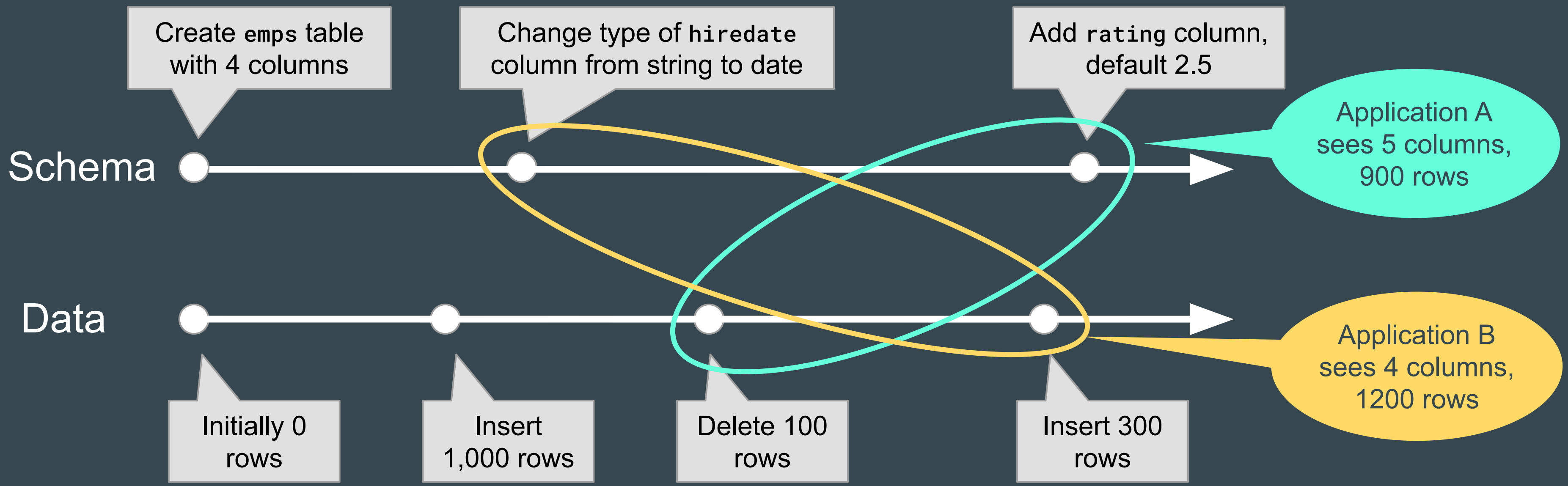
Documentation in the code

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);
```

5. Better query

```
-- 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

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
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 over p.sales}
  order DESC item
```

Morel query syntax

```
exp →
    from scan [, scan… ] [ step… ]
    [ terminalStep ]
  | exists scan [, scan… ] [ step… ]
  | forall scan [, scan… ] [ step… ]
    require condition
  | aggFn over exp[1]
  | elements[1]
  | current[2]
  | ordinal[2]
  | DESC exp
  | (* many other kinds of expression *)

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

[1] Aggregate expression and **elements** only in **compute**
[2] **current** and **ordinal** only in query

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

terminalStep →
    into exp
  | compute exp
```

How Morel improves on SQL’s SELECT

- Ordered collections
- Collections of non-records
- Queries that read like pipelines
- Function values (lambdas)
- Easily refactor query into functions
- Write your own aggregate functions
- Any expression can be correlated
- Use a query anywhere that an expression is allowed
- Define variables, intermediate results or functions anywhere
- Polymorphic types
- Sum types
- 2-valued boolean logic

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
```

Modifying data

SQL DML

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

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

-- Double the salary of all managers.
UPDATE hr.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 hr.emps
  where e.sal > 1000;

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

(* Double the salary of all managers. *)
update e in hr.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 hr.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 {hr with emps = emps4};
```

Incremental computation

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

(* Compute the updated summary table. *)
val summary2 =
  from s in hr.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 = ~e.sal})
    group s.deptno compute c = sum of c, sum_sal = sum of sum_sal
    where c <> 0;

(* Commit. *)
commit {hr 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)

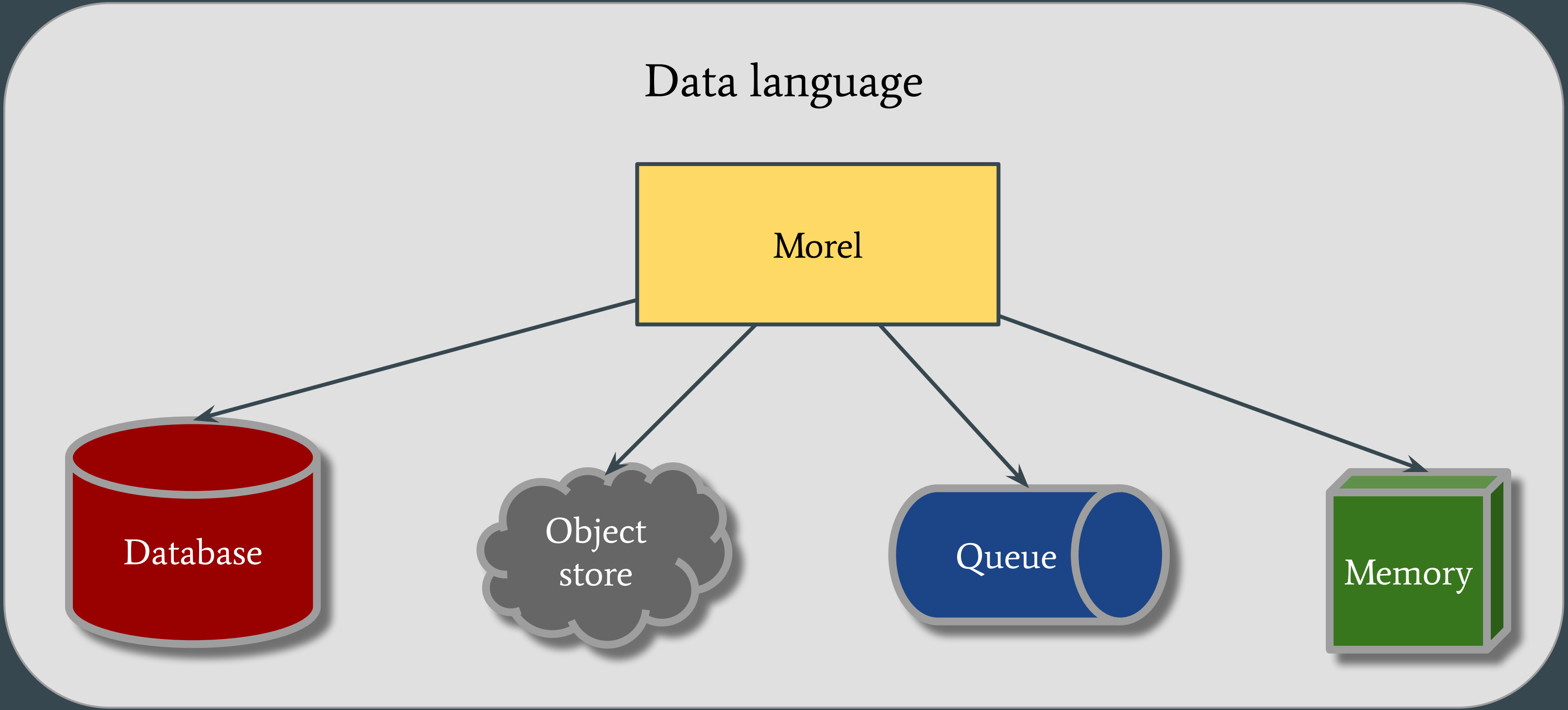
Efficiently compute deltas

Recommend materialized views

Add temporary tables for shared intermediate results

Remove unused temporary tables

Multi-query optimization [CALCITE-6188]


Data language
Morel
Database
Object
store
Queue
Memory

Morel

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

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