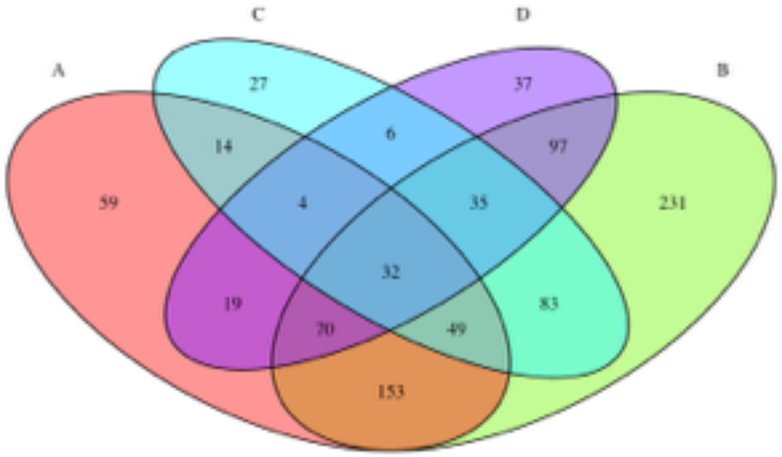


Polyalgebra

Hadoop Summit, April 14, 2016

Who

Julian Hyde @julianhyde

Apache Calcite (VP), Drill, Kylin

Mondrian OLAP

Hortonworks

Tomer Shiran @tshiran

Apache Drill

Dremio

Thanks:

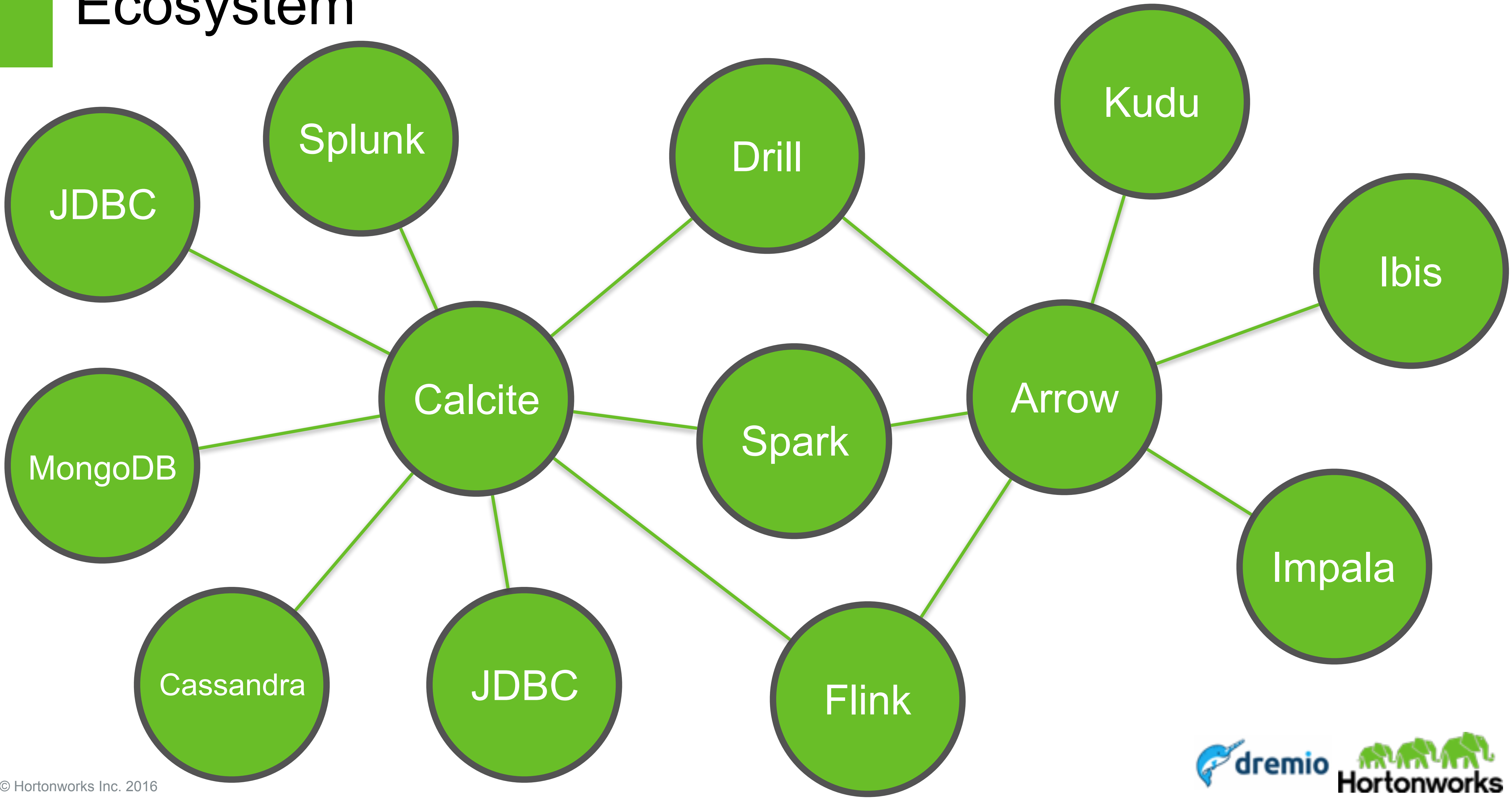
Jacques Nadeau @intjesus (Dremio/Drill)

Wes McKinney (Cloudera/Arrow)

Polyalgebra

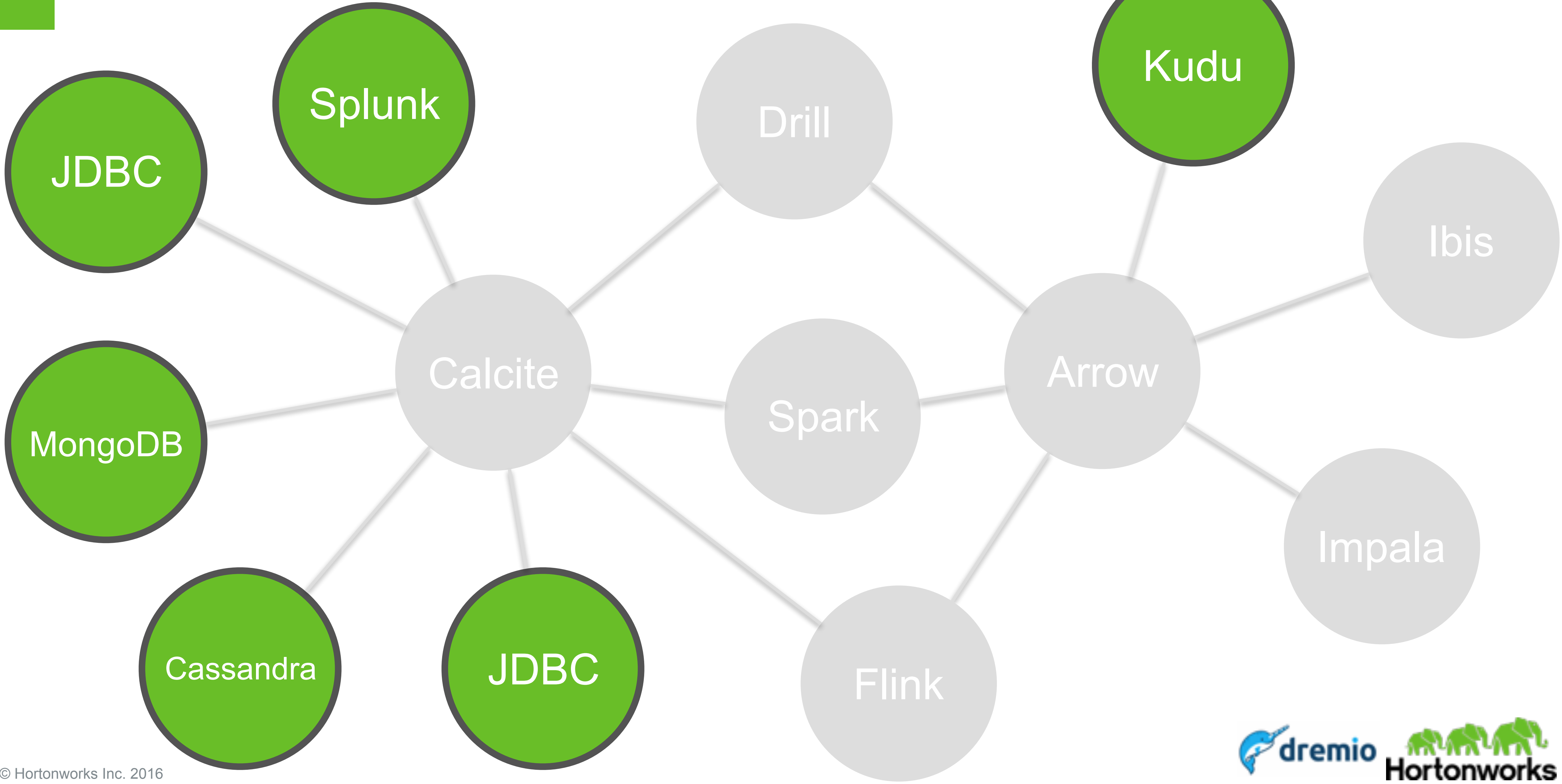
An extended form of relational algebra that encompasses work with dynamically-typed data, complex records, streaming and machine learning that allows for a single optimization space.

Ecosystem

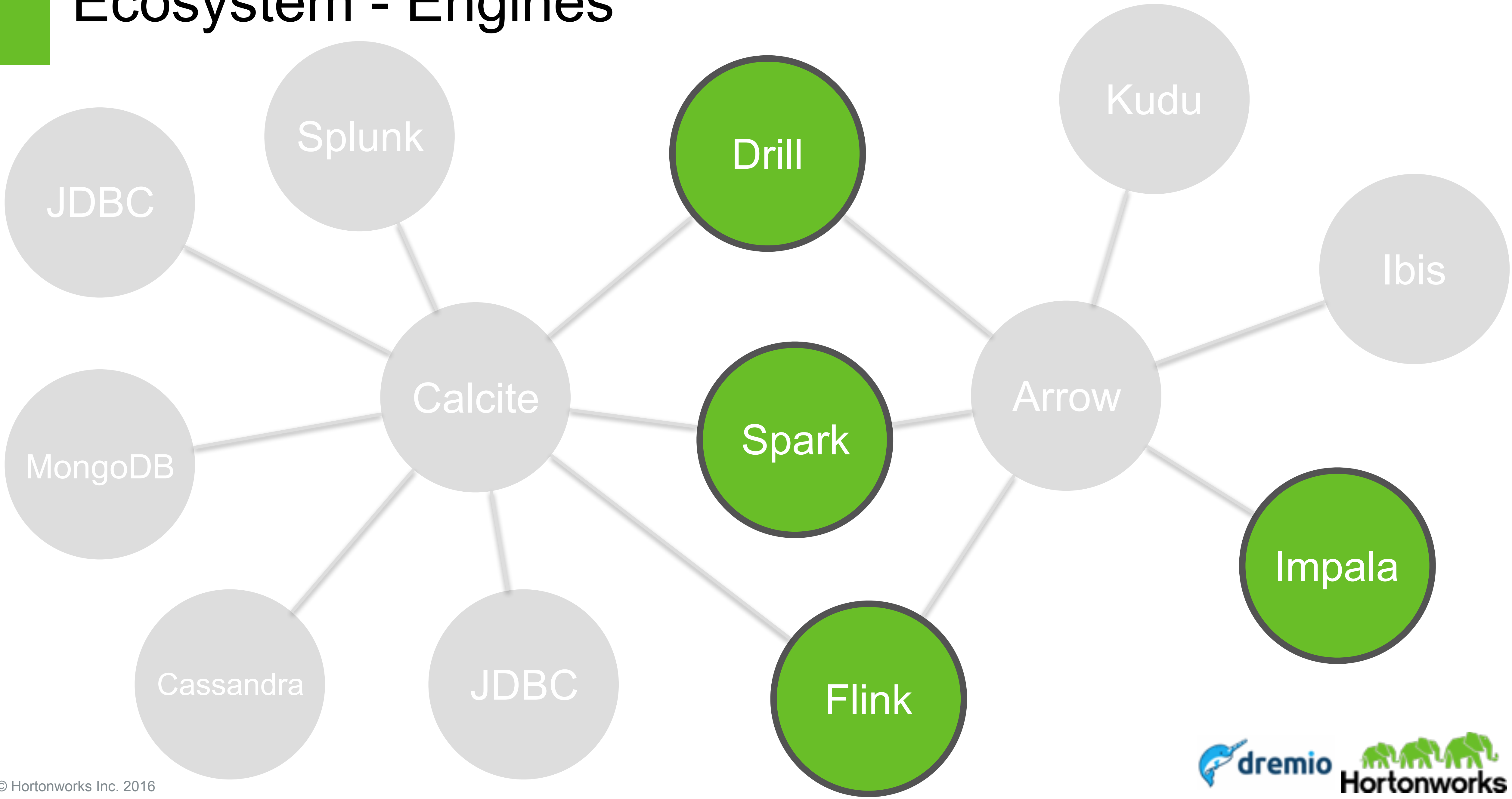


dremio Hortonworks

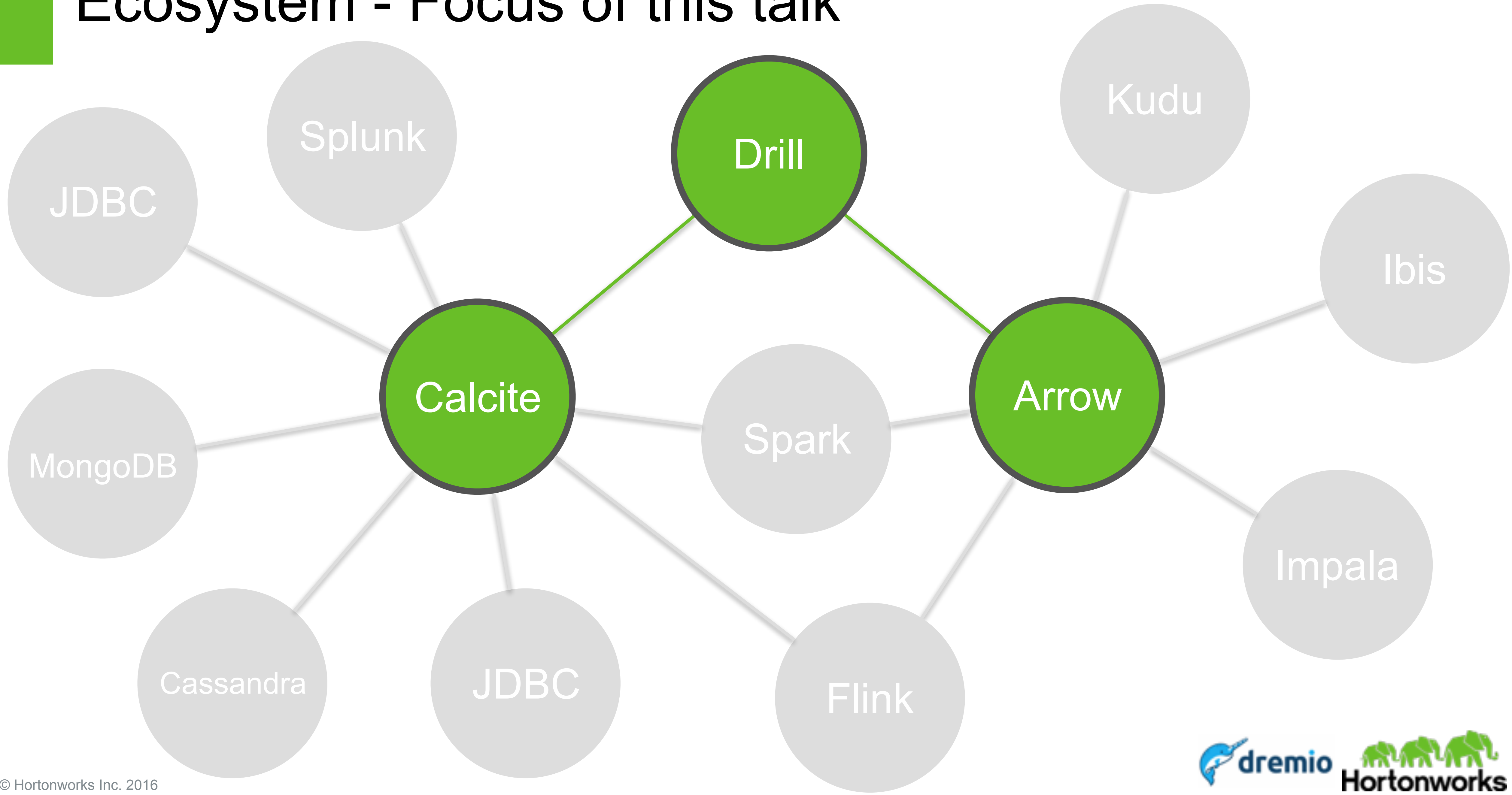
Ecosystem - data stores



Ecosystem - Engines



Ecosystem - Focus of this talk



Old world, new world

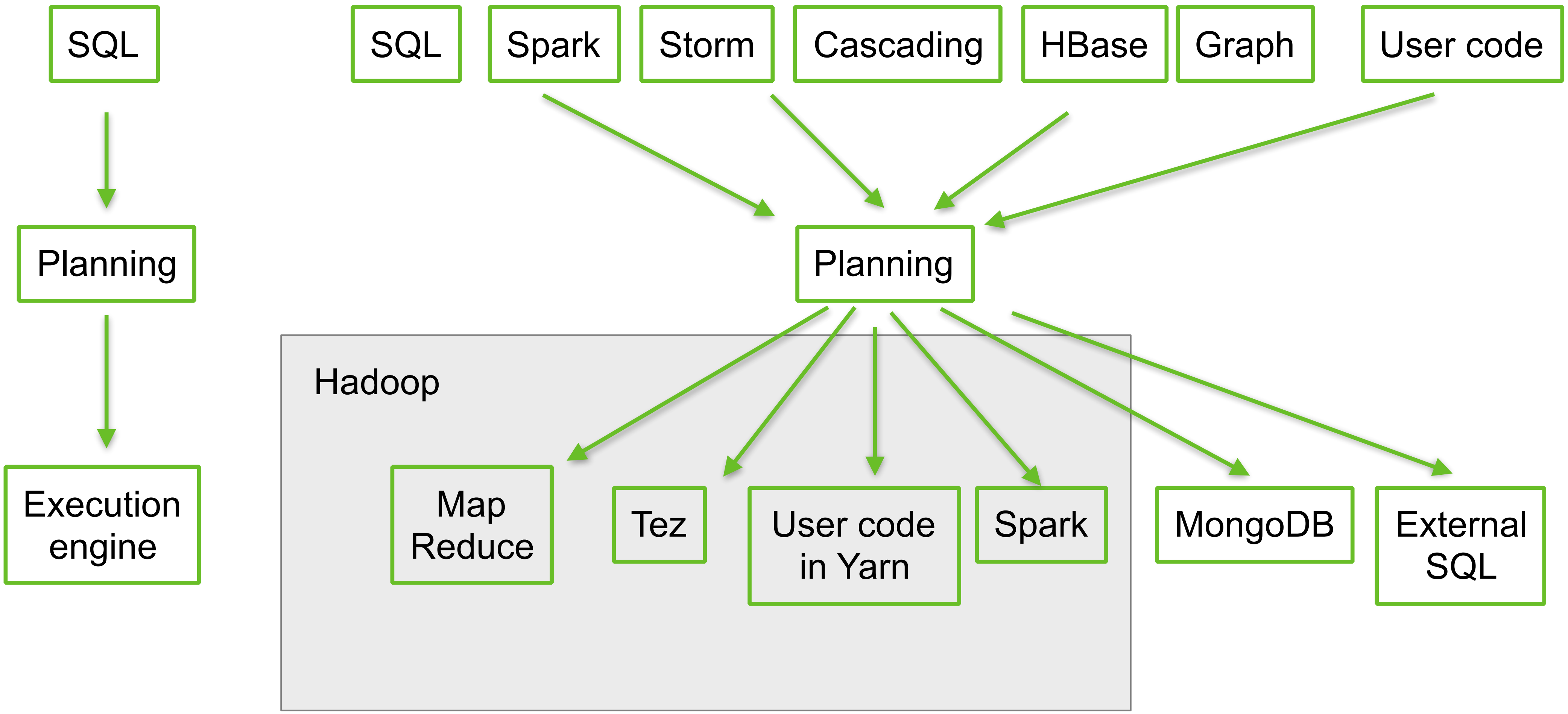
RDBMS

- Security
- Metadata
- SQL
- Query planning
- Data independence

Hadoop

- Scale
- Late schema
- Choice of front-end
- Choice of engines
- Workload: batch, interactive, streaming, ML, graph, …

Many front ends, many engines



Relational algebra

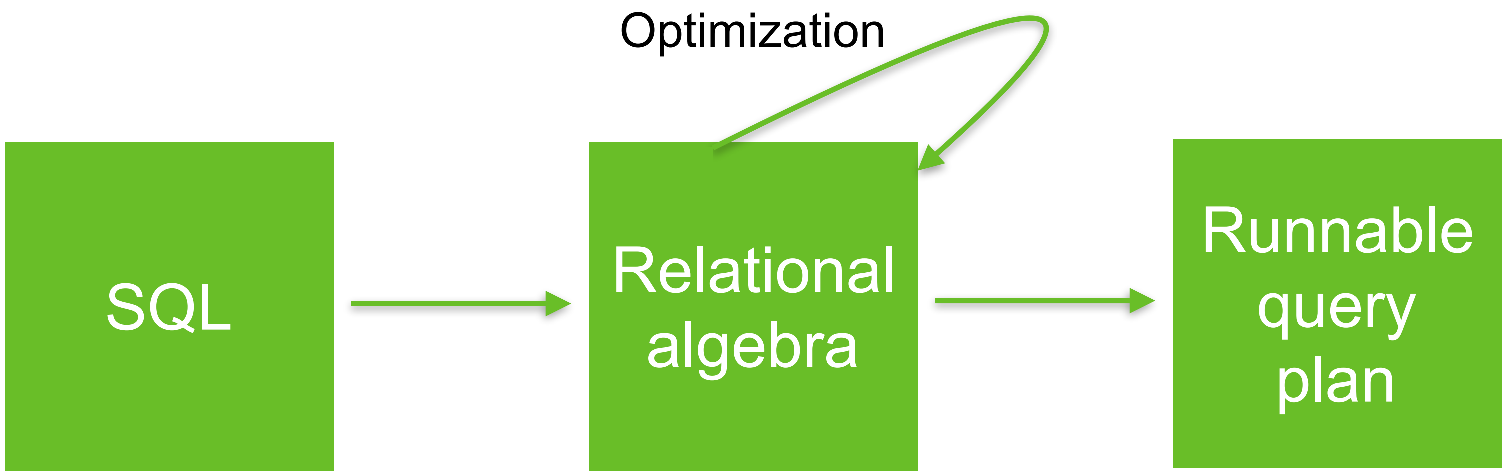
Extension to mathematical set theory

Devised by E.F. Code (IBM) in 1970

Defines the relational database

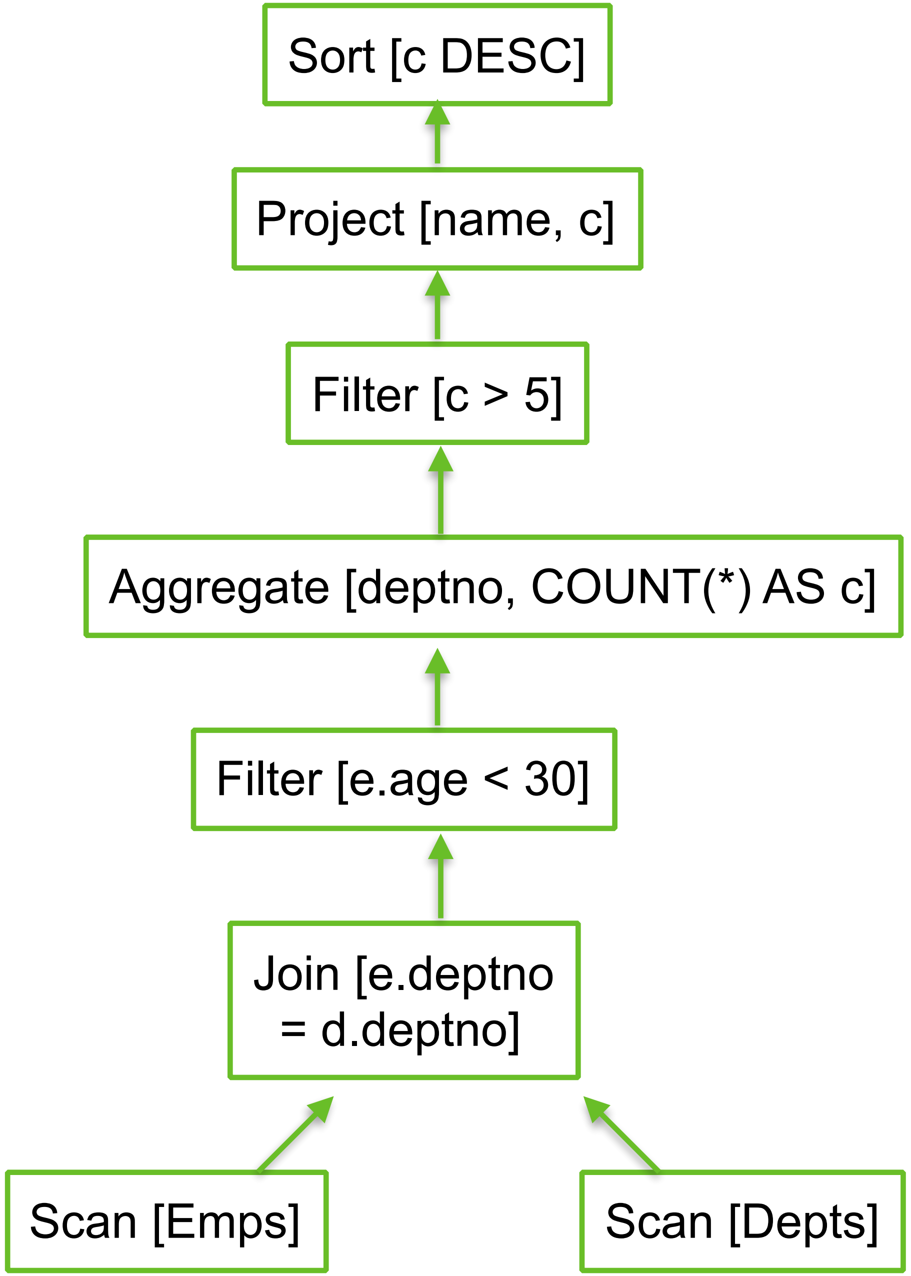
Operators: select, filter, join, sort, union, etc.

Intermediate format for query planning/optimization



Relational algebra

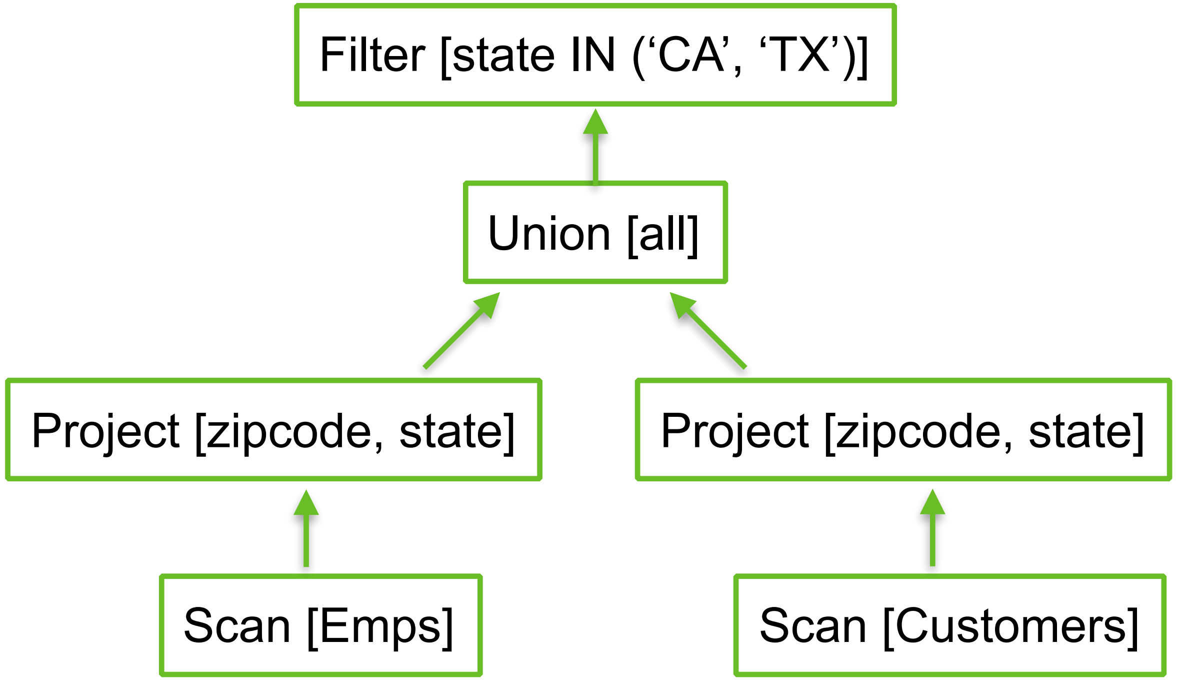
```
select d.name, COUNT(*) as c
from Emps as e
  join Depts as d
  on e.deptno = d.deptno
where e.age < 30
group by d.deptno
having count(*) > 5
order by c desc
```



(Column names are simplified. They would usually be ordinals, e.g. $0 is the first column of the left input.)

Relational algebra - Union and sub-query

```
select * from (
  select zipcode, state
  from Emps
  union all
  select zipcode, state
  from Customers)
where state in ('CA', 'TX')
```



Relational algebra - Insert and Values

```
insert into Facts
values ('Meaning of life', 42),
  ('Clever as clever', 6)
```

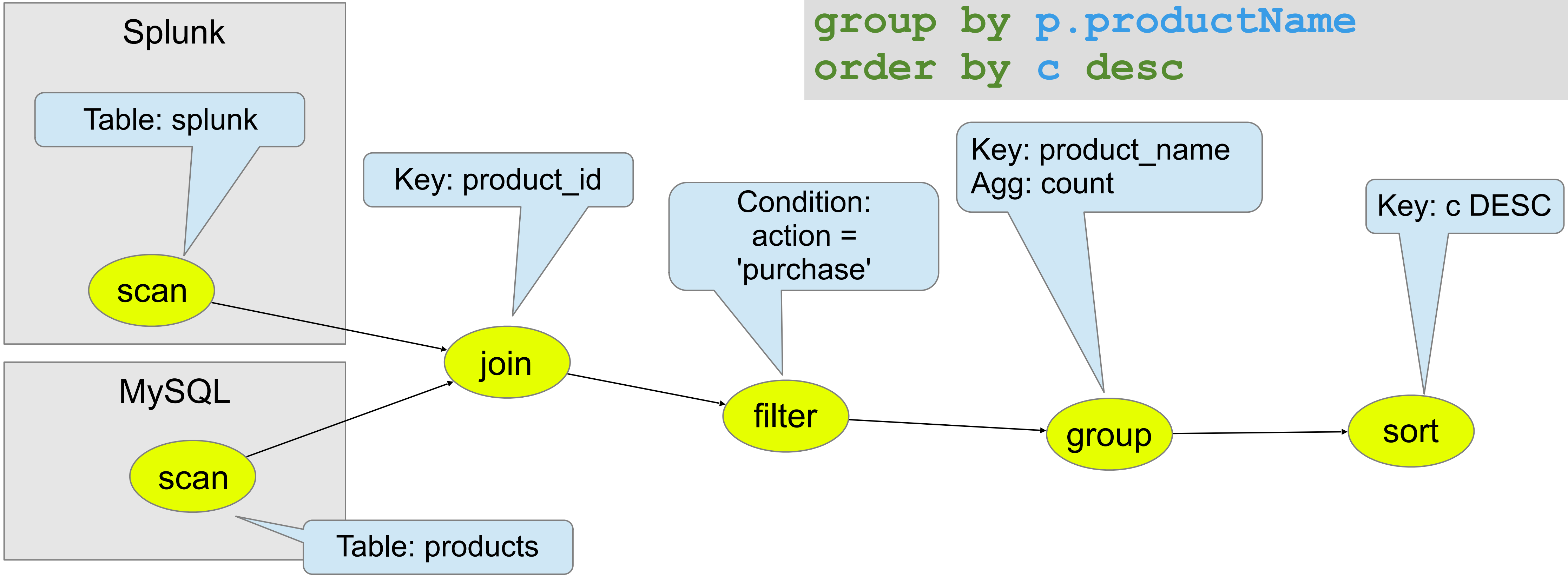
Insert [Facts]

↑

Values [['Meaning of life', 42],
['Clever as clever', 6]]

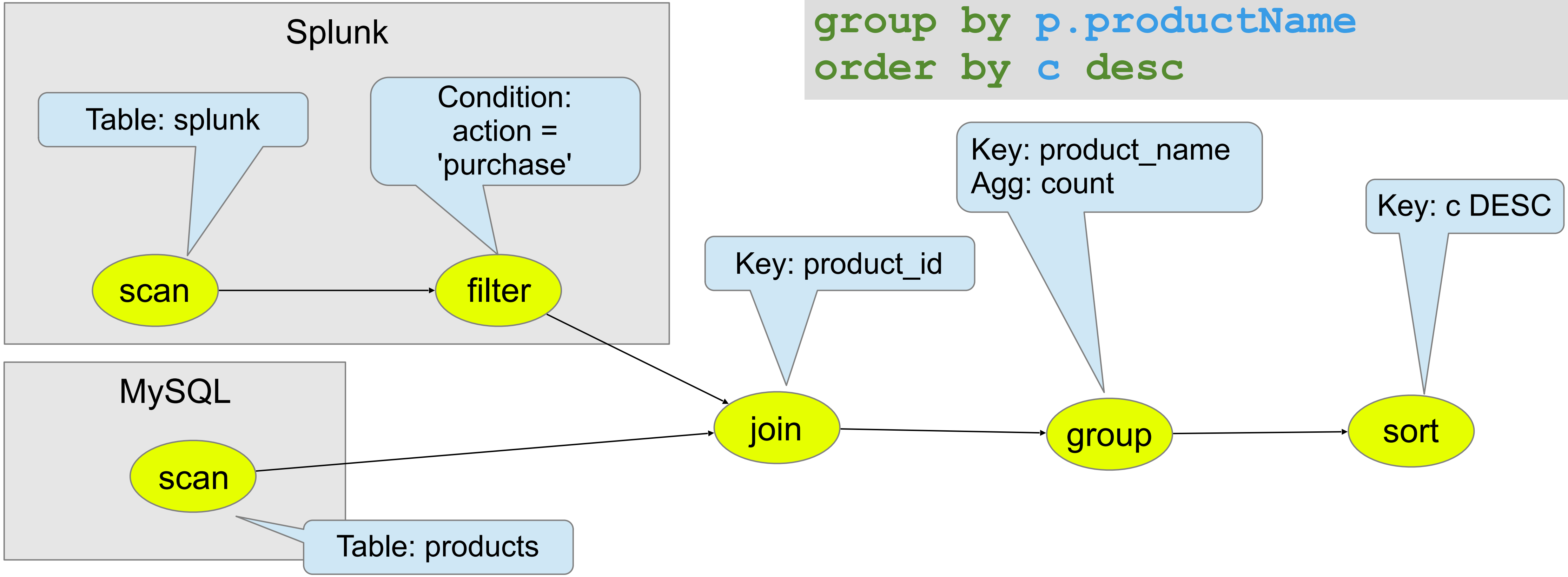
Expression tree

```
select p.productName, COUNT(*) as c
from splunk.splunk as s
     join mysql.products as p
     on s.productId = p.productId
where s.action = 'purchase'
group by p.productName
order by c desc
```



Expression tree (optimized)

```
select p.productName, COUNT(*) as c
from splunk.splunk as s
    join mysql.products as p
    on s.productId = p.productId
where s.action = 'purchase'
group by p.productName
order by c desc
```



Demo

{sqlline, apache-calcite, .csv, CsvPushProjectOntoTableRule}

Calcite – APIs and SPIs

Relational algebra

RelNode (operator)
- TableScan
- Filter
- Project
- Union
- Aggregate
- …

RelDataType (type)
RexNode (expression)
RelTrait (physical property)
- RelConvention (calling-convention)
- RelCollation (sortedness)
- TBD (bucketedness/distribution)

SQL parser

SqlNode
SqlParser
SqlValidator

Metadata

Schema
Table
Function
- TableFunction
- TableMacro

Lattice

JDBC driver

Transformation rules

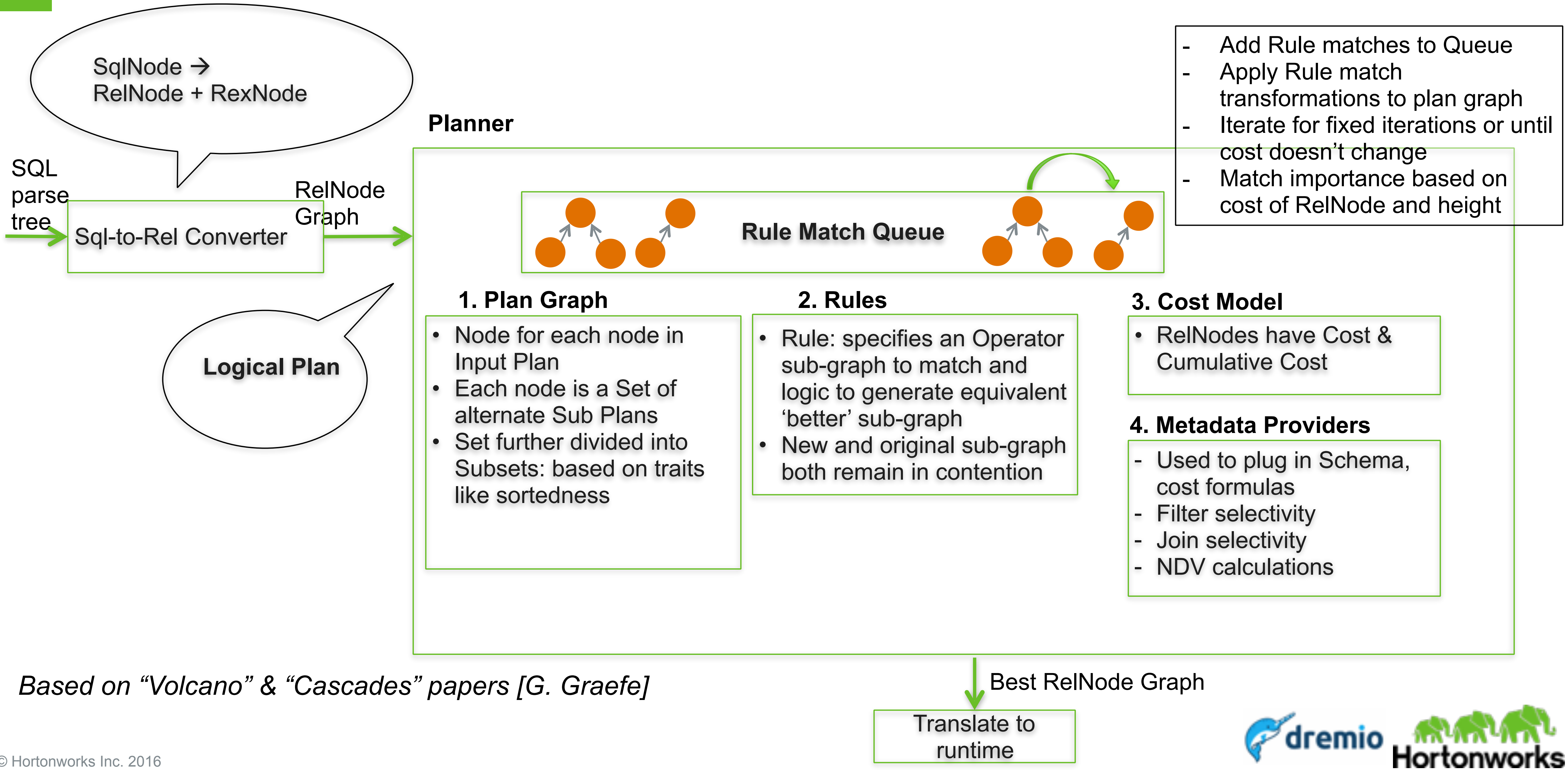
RelOptRule
- MergeFilterRule
- PushAggregateThroughUnionRule
- 100+ more

Global transformations
- Unification (materialized view)
- Column trimming
- De-correlation

Cost, statistics

RelOptCost
RelOptCostFactory
RelMetadataProvider
- RelMdColumnUniquensss
- RelMdDistinctRowCount
- RelMdSelectivity

Calcite Planning Process



Based on "Volcano" & "Cascades" papers [G. Graefe]

Algebra builder API

```
final FrameworkConfig config;
final RelBuilder builder = RelBuilder.create(config);
final RelNode node = builder
  .scan("EMP")
  .aggregate(builder.groupKey("DEPTNO"),
      builder.count(false, "C"),
      builder.sum(false, "S", builder.field("SAL")))
  .filter(
      builder.call(SqlStdOperatorTable.GREATER_THAN,
          builder.field("C"),
          builder.literal(10)))
  .build();
System.out.println(RelOptUtil.toString(node));
```

produces

LogicalFilter(condition=[>($1, 10)])
  LogicalAggregate(group=[{7}], C=[COUNT()], S=[SUM($5)])
    LogicalTableScan(table=[[scott, EMP]])

Equivalent SQL:

```
select deptno,
  COUNT(*) as c,
  sum(sal) as s
from Emp
having COUNT(*) > 10
```

Non-relational, post-relational

Non-relational stores:

- Document databases — MongoDB
- Key-value stores — HBase, Cassandra
- Graph databases — Neo4J
- Multidimensional OLAP — Microsoft Analysis, Mondrian
- Streams — Kafka, Storm
- Text, audio, video

Non-relational operators — data exploration, machine learning

Late or no schema

Complex data

Complex data, also known as nested or document-oriented data. Typically, it can be represented as JSON.

2 new operators are sufficient:

- **UNNEST**
- **COLLECT aggregate function**

```
employees: [
  {
    name: “Bob”,
     age: 48,
     pets: [
      {name: “Jim”, type: “Dog”},
      {name: “Frank”, type: “Cat”}
     ]
  }, {
    name: “Stacy",
    age: 31,
    starSign: ‘taurus’,
    pets: [
       {name: “Jack”, type: “Cat”}
     ]
  }, {
    name: “Ken”,
    age: 23
  }
]
```

UNNEST and Flatten

Flatten converts arrays of values to separate rows:

- **New record for each list item**
- **Empty lists removes record**

```
select e.name, e.age,
  flatten(e.pet)
from Employees as e
```

Flatten is actually just syntactic sugar for the UNNEST relational operator:

```
select e.name, e.age,
  row(a.name, a.type)
from Employees as e,
  unnest e.addresses as a
```

name	age	pets
Bob	48	[{name: Jim, type: dog}, {name:Frank, type: dog}]
Stacy	31	[{name: Jack, type: cat}]
Ken	23	[]

name	age	pet
Bob	48	{name: Jim, type: dog}
Bob	48	{name: Frank, type: dog}
Stacy	31	{name: Jack, type: cat}

Optimizing UNNEST

As usual, to optimize, we write planner rules.

We can push filters into the non-nested side, so we write FilterUnnestTransposeRule.

(There are many other possible rules.)

```
select e.name, a.name
from Employees as e,
  unnest e.pets as a
where e.age < 30
```

FilterUnnestTransposeRule

```
select e.name, a.street
from (
  select *
  from Employees
  where e.age < 30) as e,
unnest e.addresses as a
```

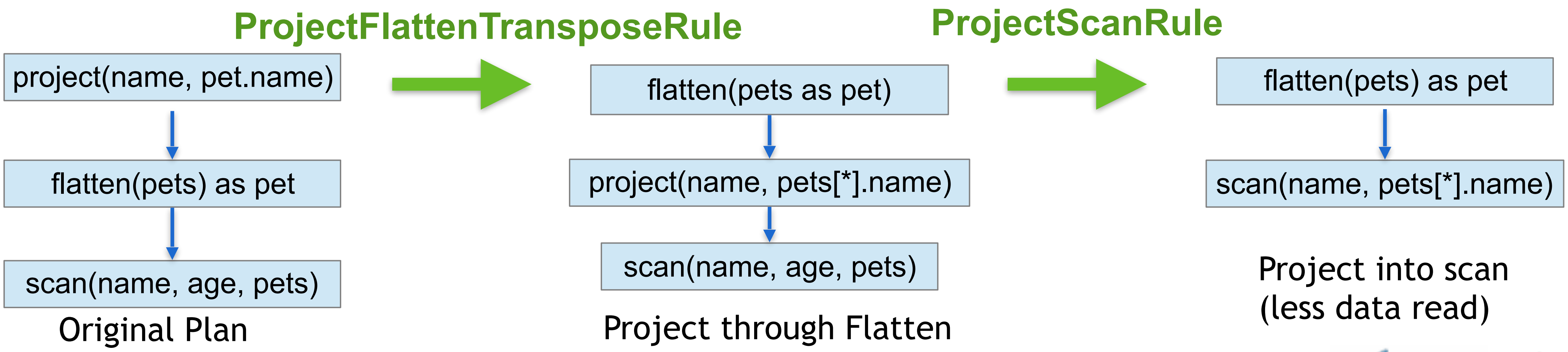
Optimizing UNNEST (2)

We can also optimize projects.

If table is stored in a column-oriented file format, this reduces disk reads significantly.

```
select e.name,
  flatten(pets).name
from Employees as e
```

- **Array wildcard projection through flatten**
- **Non-flattened column inclusion**



Late schema

Evolution:

- **Oracle: Schema before write, strongly typed SQL (like Java)**
- **Hive: Schema before query, strongly typed SQL**
- **Drill: Schema on data, weakly typed SQL (like JavaScript)**

```
select *
from Employees
```

name	age	starSign	pets
Bob	48		[{name: Jim, type: dog}, {name:Frank, type: dog}]
Stacy	31	Taurus	[{name: Jack, type: cat}]
Ken	23		[]

```
select *
from Employees
where age < 30
```

no starSign column!

name	age	pets
Ken	23	[]

Implementing schema-on-data

Expanding *

- **Early schema databases expand * at planning time, based on schema**
- **Drill expands * during query execution**
- **Each operator needs to be able to propagate column names/types as well as data**

Internally, Drill is strongly typed

- **Strong typing means efficient code**
- **JavaScript engines do this too**
- **Infer type for each batch of records**
- **Throw away generated code if a column changes type in the next batch of records**

```
select e.name
from Employees
where e.age < 30
```

→

```
select e._map["name"] as name
from Employees
where cast(e._map["age"] as integer) < 30
```

User-defined operators

A *table function* is a Java UDF that returns a relation.

- Its arguments may be relations or scalars.
- It appears in the execution plan.
- Annotations indicate whether it is safe to push filters, project through

A *table macro* is a Java function that takes a parse tree and returns a parse tree.

- Named after Lisp macros.
- It does not appear in the execution plan.
- Views (next slide) are a kind of table macro.

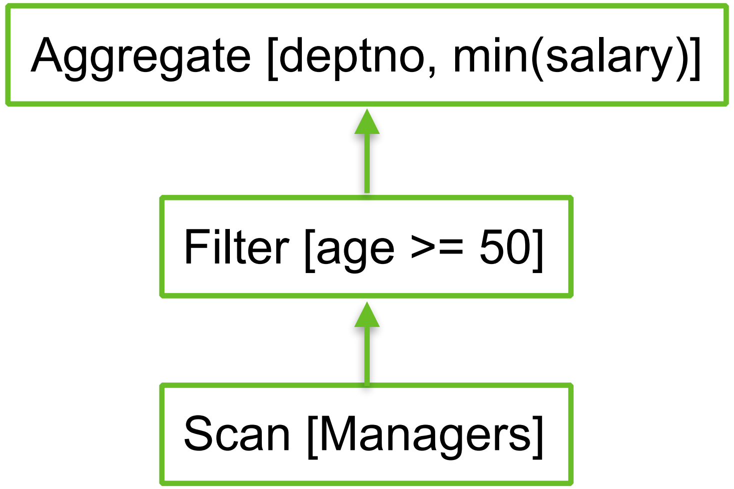
Use a table macro rather than a table function, if possible. Re-use existing optimizations.

```
select e.name
from table(
  my_sample(
    select * from Employees,
    0.15))
```

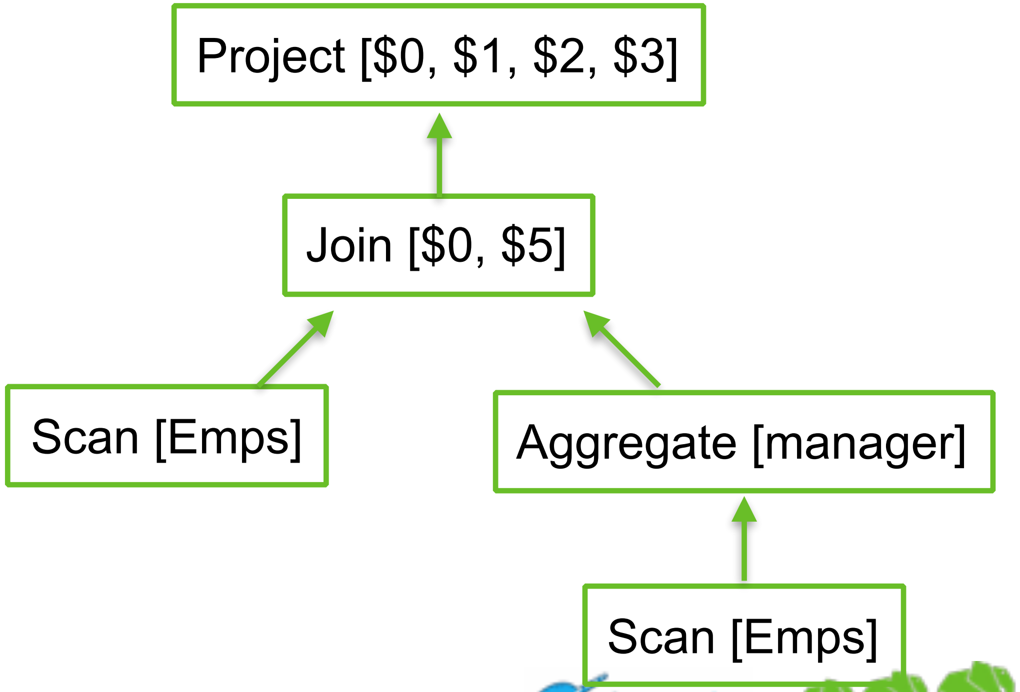
```
select e.name
from table(
  my_filter(
    select * from Employees,
    'age', '<', 30))
```

Views

```sql
select deptno, min(salary)
from Managers
where age >= 50
group by deptno
```



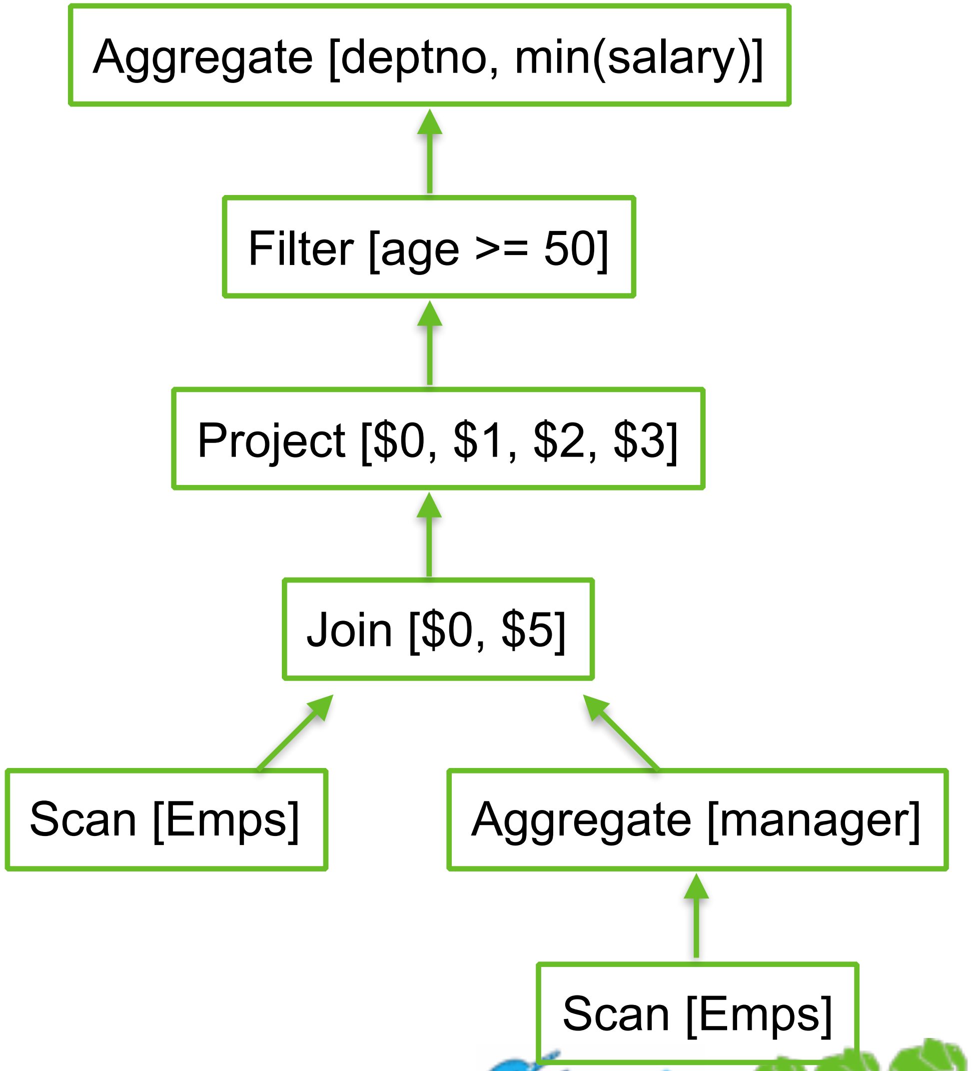

```sql
create view Managers as
select *
from Emps as e
where exists (
   select *
   from Emps as underling
   where underling.manager = e.id)
```



Views (after expansion)

```
select deptno, min(salary)
from Managers
where age >= 50
group by deptno
```

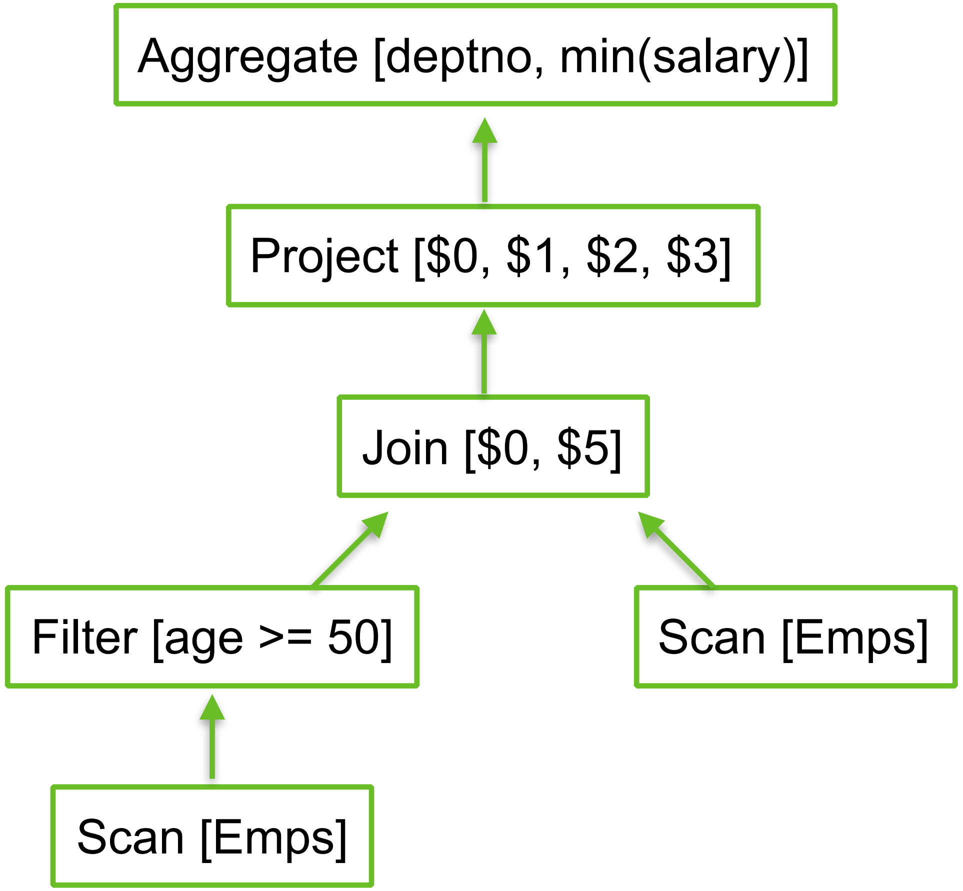
```
create view Managers as
select *
from Emps as e
where exists (
  select *
  from Emps as underling
  where underling.manager = e.id)
```



Views (after pushing down filter)

```sql
select deptno, min(salary)
from Managers
where age >= 50
group by deptno
```

```sql
create view Managers as
select *
from Emps as e
where exists (
  select *
  from Emps as underling
  where underling.manager = e.id)
```



Materialized view

```
create materialized view
  EmpSummary as
select deptno,
  gender,
  count(*) as c,
  sum(sal)
from Emps
group by deptno, gender
```

Scan [EmpSummary] = Aggregate [deptno, gender, COUNT(*), SUM(sal)] ← Scan [Emps]

```
select count(*) as c
from Emps
where deptno = 10
and gender = 'M'
```

Aggregate [COUNT(*)]

↑

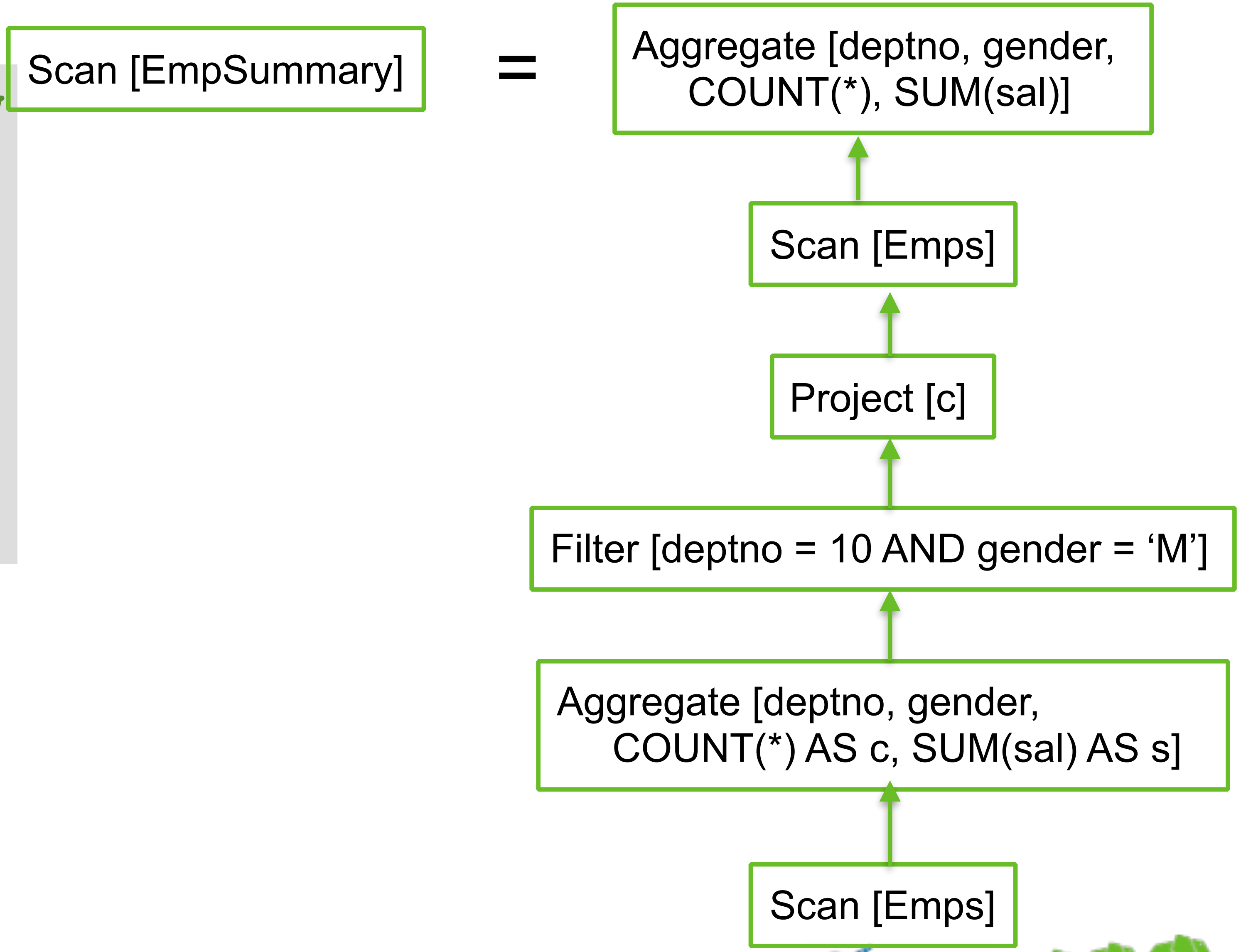
Filter [deptno = 10 AND gender = 'M']

↑

Scan [Emps]

Materialized view, step 2: Rewrite query to match

```
create materialized view
  EmpSummary as
select deptno,
  gender,
  count(*) as c,
  sum(sal)
from Emps
group by deptno, gender
```

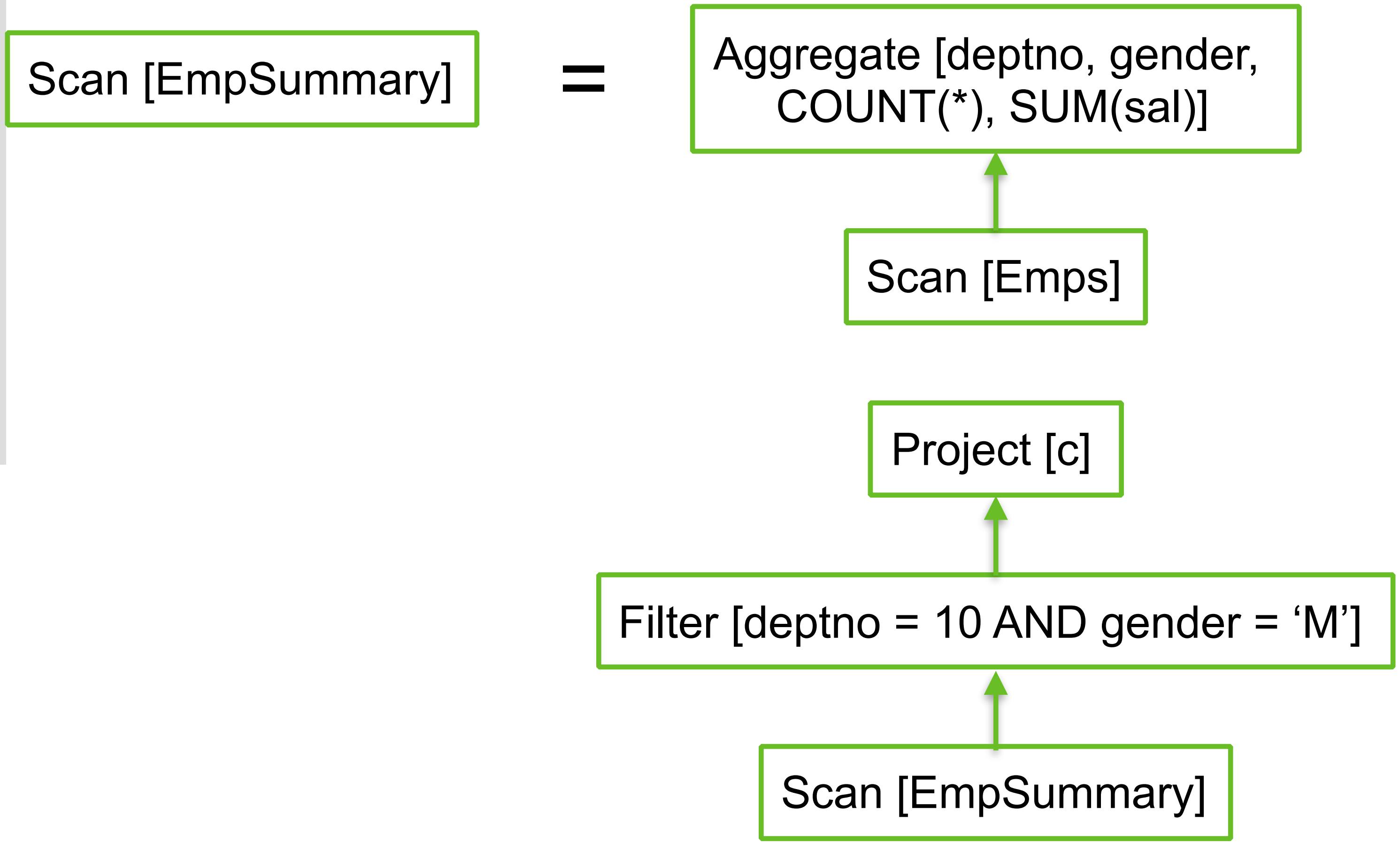



```
select count(*) as c
from Emps
where deptno = 10
and gender = 'M'
```

Materialized view, step 3: substitute table

```
create materialized view
  EmpSummary as
select deptno,
  gender,
  count(*) as c,
  sum(sal)
from Emps
group by deptno, gender
```

```
select count(*) as c
from Emps
where deptno = 10
and gender = 'M'
```



Streaming queries

Streaming queries run forever.

Stream appears in the FROM clause: `Orders`.

Without the `stream` keyword, `Orders` means the *history* of the stream (a table).

Calcite streaming SQL: in Samza, Storm, Flink.

```
select stream *
from Orders
```

```
select *
from Orders
```

Combining past and future

```
select stream *
from Orders as o
where units > (
    select avg(units)
   from Orders as h
    where h.productId = o.productId
    and h.rowtime >
        o.rowtime - interval '1' year)
```

- `Orders` is used as both stream and table
- System determines where to find the records
- Query is invalid if records are not available

Hybrid systems

Hybrid systems combine more than one data source and/or engine.

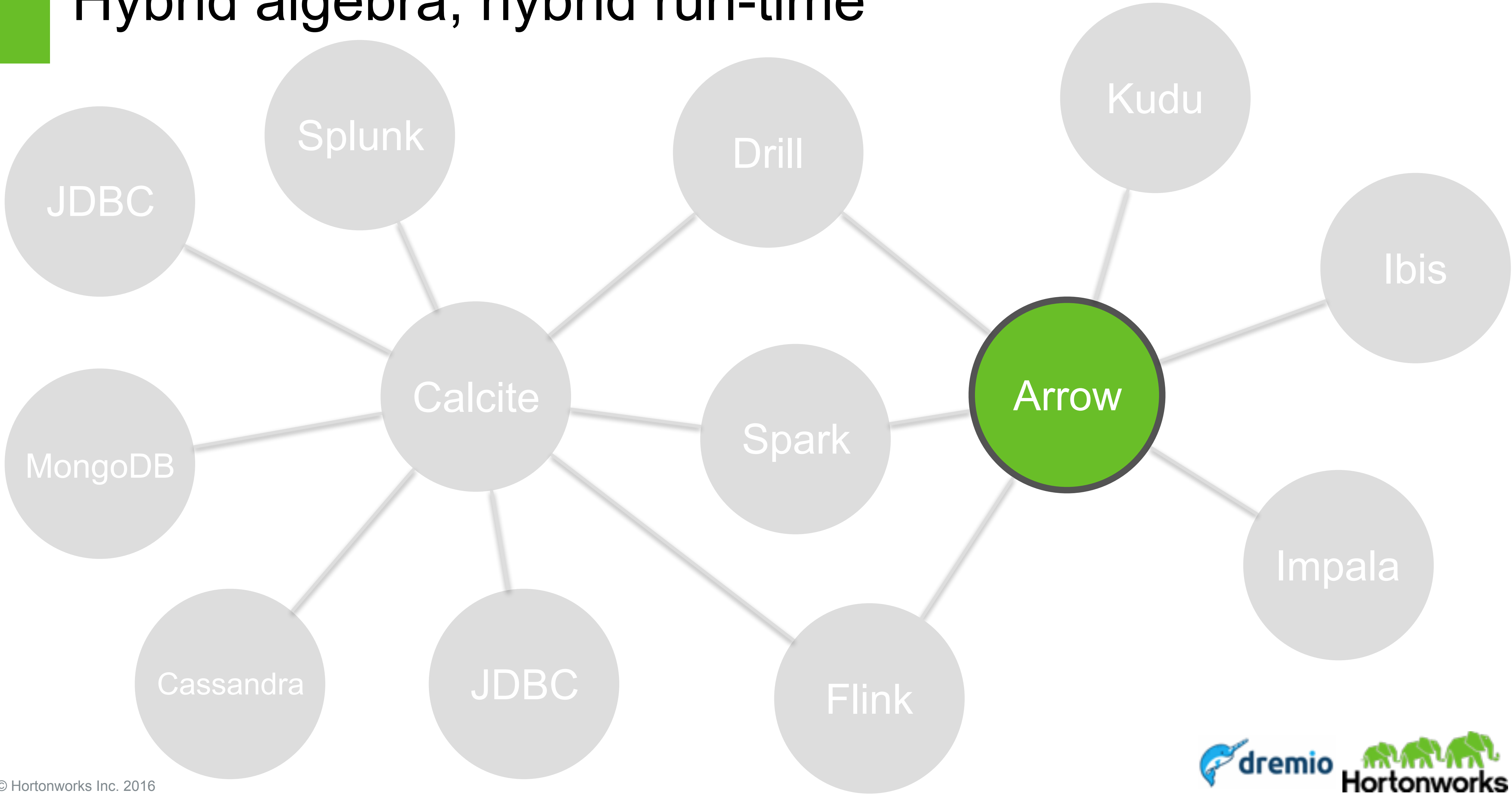
Examples:

- **Splunk join to MySQL**
- **User-defined table written in Python reading from an in-memory temporary table just created by Drill.**
- **Streaming query populating a table summarizing the last hour's activity that will be used to populate a pie chart in a web dashboard.**

Two challenges:

- **Planning the query to take advantage of each system's strengths.**
- **Efficient interchange of data at run time.**

Hybrid algebra, hybrid run-time



Arrow in a Slide

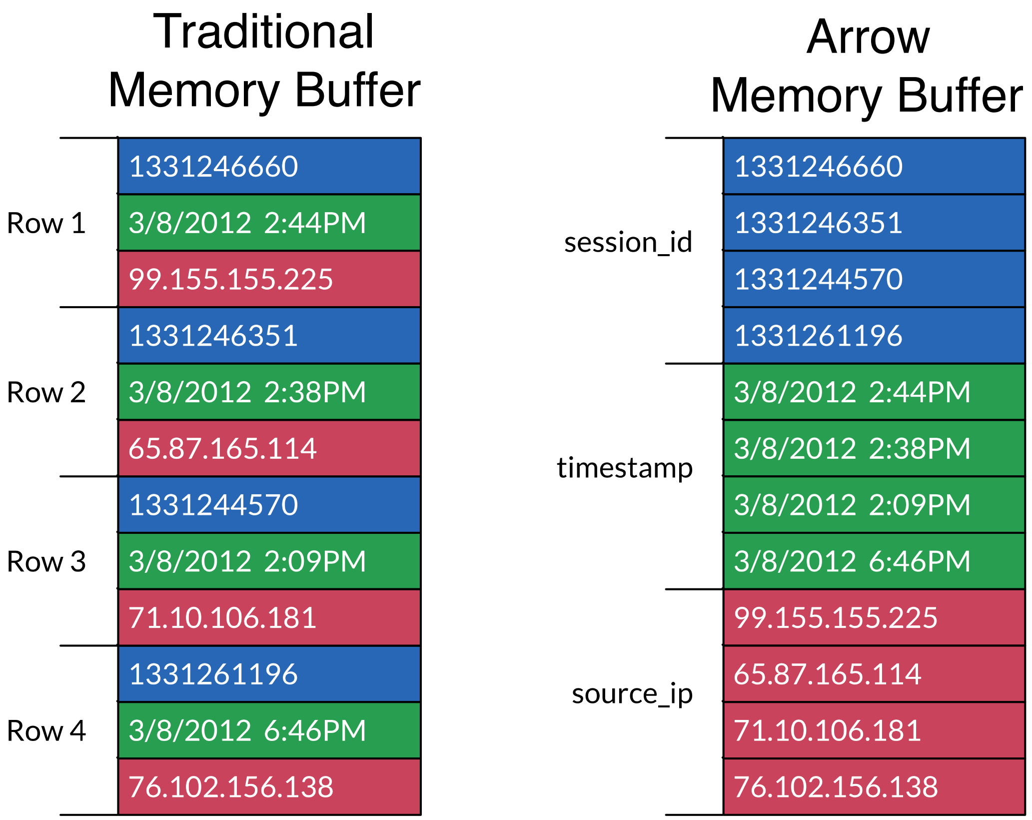
- New Top-level Apache Software Foundation project
  - Announced Feb 17, 2016
- Focused on Columnar In-Memory Analytics
  1. 10-100x speedup on many workloads
  2. Common data layer enables companies to choose best of breed systems
  3. Designed to work with any programming language
  4. Support for both relational and complex data as-is
- Developers from 13+ major open source projects involved
  - A significant % of the world's data will be processed through Arrow!

Calcite
Cassandra
Deeplearning4j
Drill
Hadoop
HBase
Ibis
Impala
Kudu
Pandas
Parquet
Phoenix
Spark
Storm
R

Focus on CPU Efficiency

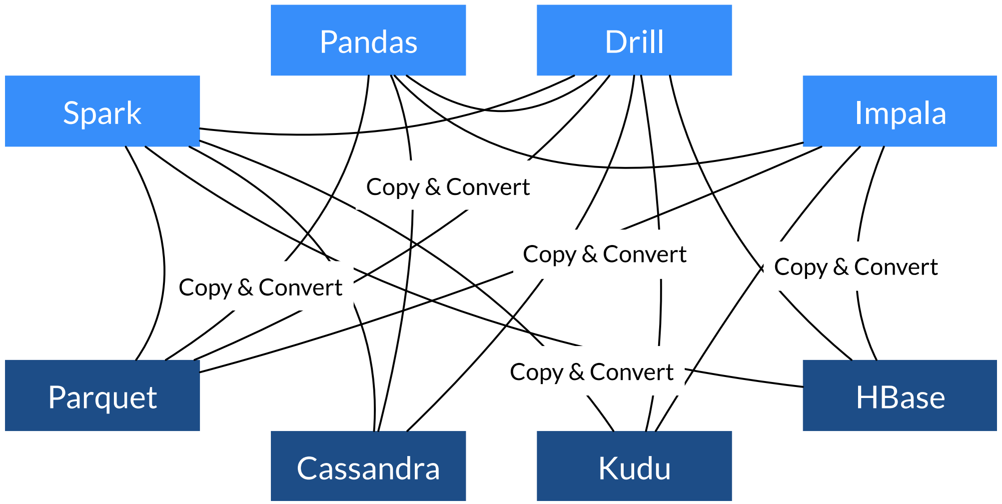
- Cache Locality
- Super-scalar & vectorized operation
- Minimal Structure Overhead
- Constant value access
  – With minimal structure overhead
- Operate directly on columnar compressed data

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



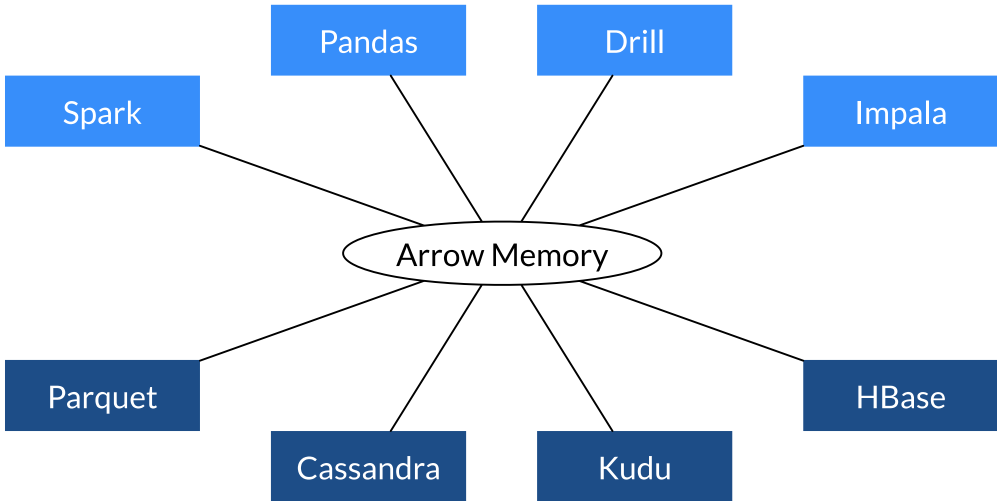
High Performance Sharing & Interchange

Today



- Each system has its own internal memory format
- 70-80% CPU wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

With Arrow



- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg, Parquet-to-Arrow reader)

Summary

- Algebra-centric approach
- Optimize by applying transformation rules
- User-defined operators (table functions, table macros, custom RelNode classes)
- Complex data
- Late-schema queries
- Streaming queries
- Calcite enables planning hybrid queries
- Arrow enables hybrid runtime

Thanks!

@julianhyde

@tshiran

@ApacheCalcite

@ApacheDrill

@ApacheArrow

Get involved:

- **http://calcite.apache.org**
- **http://drill.apache.org**
- **http://arrow.apache.org**