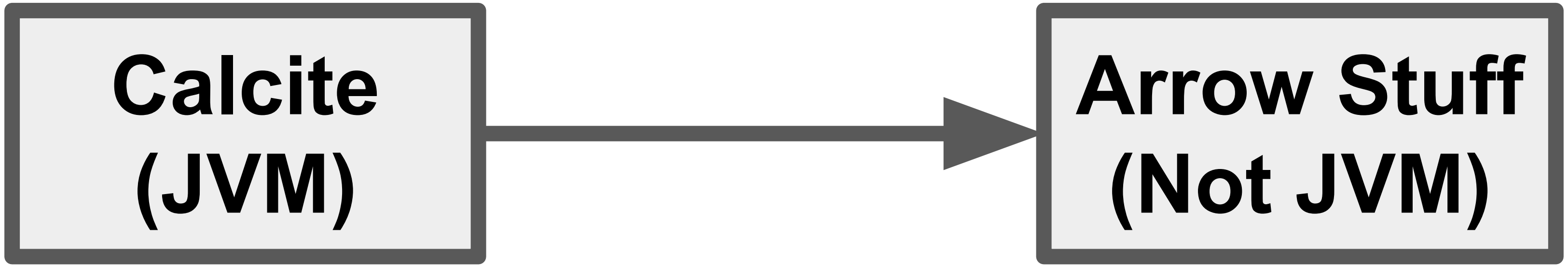
Polygraph

Adam Kennedy, Voltron Data


Calcite
(JVM)
Arrow Stuff
(Not JVM)

Motivating Challenges

Query plans are graphs

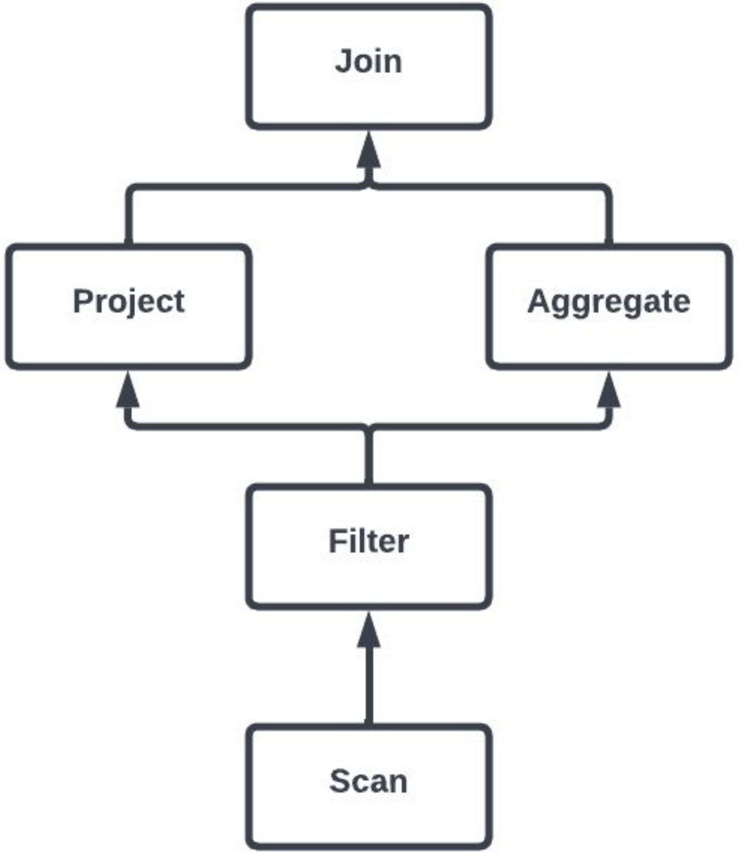
Query plans are not trees

JSON is a tree

XML is a tree(ish)

Protobuf is a tree

Substrait is a tree



Encode Plans as Graphs

Motivating Challenges

Big Data and ML plans get VERY large

10,000+ partitions and 1,000,000+ input splits

10MB+ WASM UDFs

Tables > 1MB inlined as Values

TorchData plans > 10MB

ONNX models > 500MB

Encode Plans as Columnar?

Convenient Advantages of Arrow

Arrow probably already available at run-time (for anything new)

Zero-Parse

Zero-Copy

Create and Read in Parallel

Single Node Engine → Cross-Process Shared Memory

Distributed Engine → Broadcasting Anyway (piggyback for free)

Highly Efficient IO and Compute

5x smaller than JSON uncompressed and 3x smaller compressed

IO Pruning - Only read the parts you need

Scheduler can avoid IO and parsing for 1,000,000+ file URIs

Inline (Arrow) tables can be used in place from byte range

Inline tensor weights etc can be sent to GPUs from byte range

Composable Parallel Enhancement by Adding Columns

Polygraph

Columnar Graph Representation

Polygraph Arrow Structure

Row Index	Type Enum	Inputs Integer List	Join Type Enum	Condition Protobuf Bytes
0	Output	[1]		
1	Join	[2, 3]	INNER	\$1 == \$6
2	Project	[4]		
3	Aggregate	[4]		
4	Filter	[5]		\$4 < 10
5	Scan			\$4 < 10

Polygraph Object Model

Reader

- DataFrame
- Relation[RowCount] Array

Relation

- Reader
- RowIndex Integer
- getInputs() = Reader.DataFrame.Field(Inputs).Row(RowIndex).Map(Reader.Relation[n])

Join extends Relation (x 10)

- getJoinType() = Reader.DataFrame.Field(JoinType).Row(RowIndex)

Planning Logic

Row Index	Type Enum	Inputs Integer List	Parents Integer List
0	Output	[1]	
1	Join	[2, 3]	[0]
2	Project	[4]	[1]
3	Aggregate	[4]	[1]
4	Filter	[5]	[2, 3]
5	Scan		[4]

Everything Is Naturally Indexed

Row Index	Type Enum	Column Name List Vector			Child Index
		Null Buffer	Offsets Buffer	List Data String Vector	
0	Output	NOT NULL	0	“first_name”	0
1	Scan	NOT NULL	3	“last_name”	1
			6	“age”	2
				“first_name”	3
				“last_name”	4
				“age”	5

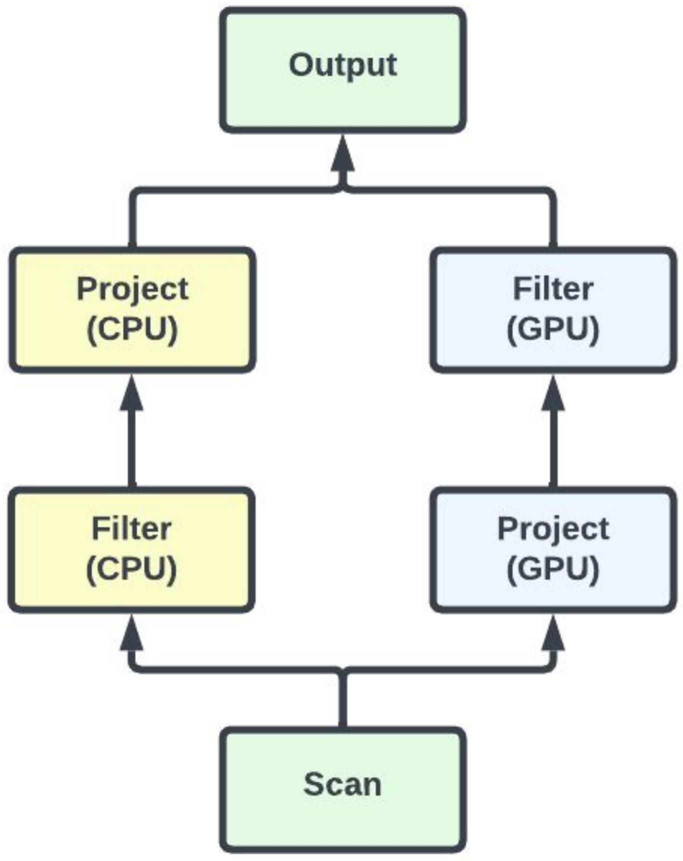
Complex “Poly” Graphs

Natural Indexing == Annotate Anything

Relation+Input Annotation for Hardware

Column Equality Graph == Reuse Memory

Trivial Property Versioning for Refectoring



Early Findings?

Faster to write, extraordinarily fast to read

Strongly typed plans, but extremely flexible “schema”

Accessor code is smaller than equivalent Protobuf

Greatly simplifies writing schedulers and run-time logic

Looks like it should round-trip for incremental planning